(4)

Productivity Engineering in the UNIX† Environment

Performance Studies of Dynamic Load Balancing in Distributed Systems

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION SPAWAR
6c. ADDRESS (City, State, and ZIP Code) Berkeley, California 94720	7b. ADDRESS (City, State, and ZIP Code) Space and Naval Warfare Systems Command Washington, DC 20363-5100
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

* PERFORMANCE STUDIES OF DYNAMIC LOAD BALANCING IN DISTRIBUTED SYSTEMS

12. PERSONAL AUTHOR(S)

* Songnian Zhou

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
technical	FROM _____ TO _____	* October 1987	* 111

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Enclosed in paper.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Performance Studies of Dynamic Load Balancing in Distributed Systems

Songnian Zhou

Report No. UCB/CSD 87/376

October 1987
PROGRES Report No. 87.6

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Performance Studies of Dynamic Load Balancing in Distributed Systems

Songnian Zhou

Performance Studies of Dynamic Load Balancing in Distributed Systems

Songnian Zhou

Abstract

Distributed systems are often characterized by uneven loads on hosts and other resources. In this thesis, the problems concerning dynamic load balancing in loosely-coupled distributed systems are studied using trace-driven simulation, implementation, and measurement. Information about job CPU and I/O demands is collected from three production systems and used as input to a simulator that includes a representative CPU scheduling policy and considers the message exchange and job transfer costs explicitly. A prototype load balancer is implemented in the Berkeley UNIX and Sun/UNIX environments, and the results of a large number of measurement experiments performed on six workstations are presented.

The quality of two families of load indices, one based on resource queue length, the other on resource utilization, is evaluated in the context of dynamic load balancing. The performances of seven algorithms using different load information exchange and job placement strategies are compared. The factors that affect load balancing performance, and the impacts of load balancing on individual hosts and on each type of job are also quantitatively investigated.

Load balancing is found to reduce significantly the mean and standard deviation of job response times, especially under heavy and/or unbalanced workload. The performance is strongly dependent upon the load index; queue-length-based indices perform better. Algorithms based on periodic or non-periodic load information exchange perform similarly. Among the periodic algorithms, the centralized ones cut down the overhead, hence scale better. The reduction of the mean response time increases with the number of hosts, but levels off beyond a few tens of hosts. Load balancing is still very effective when a large portion of the workload is immobile. All hosts, even those with light loads, benefit from load balancing. Similarly, all types of jobs see improvements in their response times, with larger jobs benefiting more. System instability is possible, but can be easily avoided. Many of the above results are likely to be of general applicability due to the excellent agreement among the simulation and measurement findings. Our implementation work shows that transparent, flexible load balancing can be achieved at low cost, without modifying the system kernel or the application programs.

Acknowledgements

Ancient poets and philosophers in China liked to regard human lives as dreams. Sometimes, dreams do come true, as in my pursuit of this degree. It would not have been possible, however, without the help and encouragement of numerous people.

I wish to express my deep gratitude to Domenico Ferrari, my research advisor, for teaching me not only how to conduct scientific research, but also how to be a researcher. During my graduate studies, Domenico has always encouraged me to explore new problems, and has always been ready to provide guidance in my wondering. Alan Jay Smith made many excellent suggestions for this dissertation. His influence on the approach and techniques used in this research is evident. Ronald Wolff put my work into a queueing network perspective, and made a number of insightful observations.

The research environment at Berkeley was highly stimulating. My numerous discussions with colleagues in the Progres group were very helpful in the formulation, solution, and presentation of the research problems. David Anderson, Riccardo Gusella, Joseph Pasquale, and Stuart Sechrest offered valuable comments on my work. Venkat Rangan and Harry Rubin made their load balancing C shell available to me as the basis of the experimental load balancer. Keith Sklower was most helpful in my implementation work in the Sun/UNIX environment.

This work was partially sponsored by the Defense Advanced Research Project Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089, and by the National Science Foundation under grant DMC-8503575. The Bell traces were made available by Will Leland of the Bell Communications Laboratory. Dennis Hall sponsored my trace collection efforts at the Lawrence Berkeley Laboratory.

Throughout the past trying years, my wife, Bing Wu, has provided constant support and encouragement.

I dedicate this dissertation to my parents, Arluo Zhou and Reiqing Gu, who taught me so many precious lessons in life. To them I owe eternal gratitude.

Table of Contents

Acknowledgements i

Table of Contents ii

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1. Load Balancing: the Motivation 1
1.2. Research Problems 3
1.3. The Main Result of the Thesis 5
1.4. Organization of the Dissertation 5

Chapter 2: Load Balancing: A Survey 6

2.1. Early work: Static Algorithms 7
2.2. Analytic Modeling and Simulation 8
2.3. Trace-driven Simulation 11
2.4. Implementation 11
2.5. Measurement 12
2.6. Load Index 13
2.7. Summary 14

Chapter 3: Trace-Driven Simulation 15

3.1. Overview 15
3.2. Experiment Design 16
3.2.1. The Job Trace 16
3.2.2. Model Structure 17
3.2.3. Overhead Cost Assumptions 19
3.2.4. Load and Performance Metrics 20
3.3. Load Balancing Algorithms 21
3.4. Simulation Results 25
3.4.1. Comparison of the Algorithms 25

3.4.2. Effect of System Load 30
3.4.3. Parameter Sensitivity and Adaptive Load Balancing 30
3.4.4. Effect of Immobile Jobs 36
3.4.5. Effect of Overhead Costs 36
3.4.6. Impact on Individual Hosts 40
3.4.7. Impact on Each Job Class 45
3.4.8. System Instability 48
3. 5.Summary 51

Chapter 4: Measurement Studies 53

4.1. Overview 53
4.2. Design and Implementation 54
4.2.1. System Basics 55
4.2.2. Algorithms 58
4.2.3. Overhead Assessment 59
4.3. Experiment Design 60
4.3.1. Performance Index 60
4.3.2. Experimental Factors 61
4.3.3. Load Index 62
4.3.4. Workload 63
4.4. Comparison of Load Indices 65
4.4.1. Canonical Workload 65
4.4.2. Moderate, Balanced Workload 67
4.5. Comparison of Algorithms 68
4.5.1. Basic Comparisons 68
4.5.2. Adjustable Parameters 69
4.6. Performance under Different Workloads 72
4.6.1. Different Intensities 72
4.6.2. Different Mobilities 75
4.7. Effects on Individual Hosts and Job Types 76
4.7.1. Effects on Individual Hosts 77
4.7.2. Effects on Each Type of Jobs 81
4.8. Summary 81

Chapter 5: Load Balancing in Other Environments 84

5.1. Overview 84
5.2. Workload Characterization 84
5.3. Simulations Driven by the Bell Traces 88
5.3.1. Comparison of the Algorithms 89
5.3.2. Effect of Immobile Jobs 91

5.3.3. Impact on Individual Hosts 91
5.3.4. Impact on Each Job Class 94
5.4. Simulations Driven by the LBL Traces 95
5.5. Summary 96

Chapter 6: Final Remarks 99

6.1. Conclusions 99
6.2. Future Work 101

Bibliography 104

List of Figures

Figure 3.1. Structure of the system model used in the simulation 19
Figure 3.2a. Average response times with different system sizes (7-49). 26
Figure 3.2b. Average response times with different system sizes (1-7). 27
Figure 3.3. Average response times under different load levels. 31
Figure 3.4. Effect of adjustable parameters, GLOBAL. 32
Figure 3.5. Effect of adjustable parameters, LOWEST 33
Figure 3.5a Effect of adjustable parameters, RESERVE 34
Figure 3.6. Effect of immobile jobs 37
Figure 3.7. Effect of message exchange costs 38
Figure 3.8. Effect of job transfer costs 39
Figure 3.9. Host CPU utilizations of a 14 host System 41
Figure 3.10. Mean response times of individual hosts 42
Figure 3.11. Standard deviation of response times of individual hosts 43
Figure 3.12. Average queue lengths of individual hosts 44
Figure 3.13. The loads on sample hosts as a function of time 45
Figure 3.14. Average response times for several classes of jobs 46
Figure 3.15. Average response times for several sizes of jobs 47
Figure 3.16. Percentage of wrong job placements 49
Figure 3.17. Distribution of the number of hosts with the least load 51

Figure 4.1. Structure of the load balancer 57
Figure 4.2. Mean response time as a function of P 70
Figure 4.3. Mean response time as a function of T_l 71
Figure 4.4. Mean response time as a function of L_p 73
Figure 4.5. The influence of immobile jobs, mean response time 76
Figure 4.6. The influence of immobile jobs: standard deviation 77
Figure 4.7. Loads on hosts, with various immobility factors 78
Figure 4.8a. The load on each host as a function of time, NoLB 79
Figure 4.8b. The load on each host as a function of time, GLOBAL 80

Figure 5.1. Distributions of jobs by their CPU demands 86
Figure 5.2. Cumulative distributions of job CPU time 88
Figure 5.3. Distributions of jobs by their file I/O 89
Figure 5.4. Average response times with different system sizes (Bell) 90

Figure 5.5. Effect of immobile jobs (Bell) 92
Figure 5.6. Host CPU utilizations of a 14 host system (Bell) 93
Figure 5.7. Mean response times of individual hosts (Bell) 93
Figure 5.8. Standard deviation of response times of individual hosts (Bell) ... 93
Figure 5.9. Average queue lengths of individual hosts (Bell) 93
Figure 5.10. Average response times for several classes of jobs (Bell) 94
Figure 5.11. Average response times for several sizes of jobs (Bell) 95
Figure 5.12. Average response times with different system sizes (LBL) 97

List of Tables

Table 1.1. A snapshot of load condition at Berkeley 2

Table 3.1. Basic statistics of the Berkeley data 18
Table 3.2. Symbols used in Chapters 3 and 4, and their meanings 24
Table 3.3. Optimal parameter values under different system load levels 35

Table 4.1. Load balancing overhead measurements 60
Table 4.2. Experimental factors and their levels 61
Table 4.3. Commands used in scripts and their eligibilities 64
Table 4.4. Characterization of the workload levels 64
Table 4.5. Measured performance with various indices (<2H, 2M, 2L>) 66
Table 4.6. Measured performance with various indices (<6M>) 67
Table 4.7. Performance of the algorithms 68
Table 4.8. Performance of five hosts with heavy loads 74
Table 4.9. Performance of six hosts with moderate loads 74
Table 4.10. Performance of six hosts with light loads 74
Table 4.11. Average response time of each command type 82

Table 5.1. Basic Statistics of the Berkeley Data 87
Table 5.2. Basic Statistics of the Bell Data 87
Table 5.3. Basic Statistics of the LBL Data 87

Chapter 1

Introduction

1.1. Load Balancing: the Motivation

Distributed computer systems are becoming increasingly available because of the rapid decrease in hardware costs and the advances in computer networking technologies. An important advantage of distributed systems is the potential for resource sharing, to provide the users with a rich collection of resources that are usually unavailable or highly contended for in stand-alone systems. Examples of sharable resources are files, computing power, and printers. It is frequently observed that, in a computing environment with a number of hosts connected by networks, the hosts are often loaded very differently. Even if the hosts are evenly loaded over long periods, such as half an hour or more, the instantaneous loads are likely to be fluctuating constantly†. Table 1.1 shows a snapshot of the loading conditions of some of the hosts in the EECS Department of the University of California, Berkeley. The hosts include VAX-11 780, 750, 785, and 8600 models running Berkeley UNIX 4.3 BSD‡. The numbers on the right provide a rough measure of the numbers of active jobs on the hosts, averaged over the last 1, 5 and 15 minutes (the UNIX *load averages*). As can be easily observed, some hosts are idle or almost idle, while others have many jobs queued up and competing for resources.

Livny and Melman pointed out, using simple queueing analysis and assuming job arrivals following a Poisson pattern, that in a multi-host system the probability of one of the hosts being idle while some other host has multiple jobs queued up can be very high [Livny85]. Such imbalances in system load suggest

---

† Such observations, of course, are dependent on the system and the applications being run. For instance, in a main-frame batch data processing environment, the loads might be even over long periods of time. However, in a workstation-rich environment, which is becoming more and more popular, the probability of a majority of the stations being idle or almost idle is very high [Theimer85].

‡ VAX is a trademark of Digital Equipment Corporation. UNIX is a trademark of AT&T Bell Laboratories.

Table 1.1. A Snapshot of Load Condition at Berkeley.

March 13th (Friday), 11:30 am, 1987, at Berkeley . . .

% ruptime

name	status	d+h:m	users on	load averages
cad	up	5+14:56,	12 users,	load 4.44, 5.21, 5.87
cartan	up	1+18:57,	4 users,	load 2.82, 2.48, 2.80
cgl	up	21+13:16,	10 users,	load 0.95, 0.94, 0.96
cogsci	up	35+13:08,	5 users,	load 0.26, 0.29, 0.48
cory	up	2+02:44,	20 users	load 7.11, 7.74, 8.11
csgw	up	2+22:21,	0 users,	load 0.00, 0.00, 0.01
degas	down	0:55		
ernie	up	3:23,	55 users,	load 10.88, 11.89, 11.37
eros	up	13:32,	29 users,	load 8.41, 6.70, 6.78
esvax	up	4+04:49,	27 users,	load 1.69, 1.30, 1.08
ji	up	77+19:55,	11 users,	load 0.59, 0.72, 0.66
matisse	up	1+17:28,	0 users,	load 0.00, 0.00, 0.01
medea	up	2:06,	9 users,	load 5.32, 5.02, 4.54
miro	up	27+01:41,	4 users,	load 2.56, 3.00, 3.34
monet	up	10+17:07,	3 users,	load 0.00, 0.04, 0.08
nova	up	34+20:07,	0 users,	load 0.11, 0.10, 0.10
okeeffe	up	4+23:05,	7 users,	load 0.00, 0.01, 0.01
renoir	down	0:54		
ucbarpa	up	6+00:31,	28 users,	load 8.40, 9.13, 9.05
ucbvax	up	8+02:02,	1 user,	load 2.18, 2.31, 2.62
vangogh	up	2:55,	1 user,	load 0.12, 0.11, 0.22

that performance can be improved by transferring jobs from the currently heavily loaded hosts to the lightly loaded ones. This form of computing power sharing, with the purpose of improving the performance of a distributed system by redistributing the workload among the available hosts, is commonly called *load balancing*, or *load sharing*†.

---

† The term load balancing has sometimes been used to imply the objective of equalizing the loads of the hosts, whereas load sharing simply means a redistribution of the workload. We will use the term load balancing in the rest of this dissertation, but without the stronger connotation. The potential of load balancing in improving performance has also been pointed out by Wolff [Wolff87].

1.2. Research Problems

The sharing of computing resources in a centralized system is implicit --- all user jobs are submitted to the single CPU, and they share the CPU according to some CPU scheduling policy. In a distributed system, however, jobs usually arrive at the various hosts independently, causing uneven loads on the hosts. To balance the loads effectively, a number of research problems have to be solved. Listed below are some of them.

1) *How much performance improvement can be expected from load balancing?*

   Since load and job information may need to be exchanged among the hosts to facilitate job transfer decisions, and jobs need to be transferred, load balancing will cause overhead. Thus, there exists a tradeoff between cost and benefit. The answer to this question depends on a number of factors, such as the system overhead, the workload, and the algorithm used for load balancing.

2) *What algorithm to use?*

   Different strategies, or algorithms, can be used to exchange load and job information, to decide which jobs should be transferred to remote hosts for execution, and to which hosts such jobs should be sent. Different load balancing algorithms are likely to produce varying performance. The quality of an algorithm is also dependent on the computing environment. It is impossible to have one algorithm that is the "best" in all circumstances. Hence, the characteristics of possible algorithms need to be understood.

3) *What load index to use?*

   Generally speaking, jobs should be transferred from hosts with heavy load to those with light load. However, this brings up the more fundamental issue of how the load of a host should be measured, or what *load index* should be used.

4) *How does the system's workload affect load balancing performance?*

   Conceivably, the levels of congestion on the resources, the degree of the workload unbalances, and those jobs that have to be executed locally all have strong influence on performance. Moreover, the distributions of the jobs' resource consumptions, and the job arrival pattern probably also have a bearing on the performance. Thus, finding that load balancing is able to improve performance on one workload does not mean that it is generally beneficial. The interactions between workload and performance under load balancing should be studied.

5) *What is the impact of load balancing on system behavior?*

   While the performance of hosts with heavy load may be improved by load balancing, this may be achieved at the expense of those with light load. A

similar situation may arise among different types of jobs. Such uneven effects may be undesirable. In a distributed system, the system state information is also scattered at many locations. Hence, the information used in load balancing is likely to be out of date, and there exists the possibility of system instability when jobs are being transferred around.

In this dissertation, we seek answers to the above questions, and conduct experiments to quantitatively assess the performance potential of load balancing.

Substantial work has been done on load balancing in the past years, taking on a variety of forms. The general problem may be studied in different types of computing environments, using different strategies, and at different levels. The system may be *loosely-coupled*, with a number of functionally complete computers connected by one or more networks through which messages may be transmitted and remote resources accessed, or may be *tightly-coupled*, with several CPU-memory combinations connected by a bus to shared memory and secondary storage. The resources to be shared may be of the same type and capacity (*homogeneous system*), or of different types and/or capacities (*heterogeneous system*). The algorithm used for load balancing may require no information, or only information about the individual jobs (*static* algorithm), or may make job transfer decisions based on the current load situation (*dynamic* algorithm). The algorithm may treat the resources as cooperating equals (*distributed* algorithm), or employ some centralized mechanism to exchange load information and/or make job placement decisions (*centralized* algorithm). The transfer of a job may be initiated by the originating host (*source-initiative* algorithm), or by the target host (*server-initiative* algorithm). The unit of execution that is to be redistributed may range from complete jobs submitted by users, or individual processes, or even smaller program modules. The units may also be components of parallel computations with specific communication requirements. Finally, the transfer of a job may be restricted to be done prior to the start of its execution (*initial job placement*), or may also be allowed during its execution (*process migration*).

Obviously, with this myriad of available choices of the subdomains for load balancing research, it is impossible to study them all in one single research effort. To make the work manageable, we limited the scope of our research to loosely-coupled distributed systems. The jobs were assumed to be sequential jobs as opposed to parallel programs with multiple modules executing on a number of hosts. Job transfers were restricted to be at the time of each job's arrival. Once a job starts executing on a host, it will not be considered for migration. We believe that the problem domain we address in this research represents the basic one in load balancing, and the results from this research may be used as the foundations for future explorations into other areas. At the end of this dissertation, we will discuss the extensibility of our work, and point out directions for future research.

1.3. The Main Result of the Thesis

While a large number of issues are studied, and the results are multi-dimensional, the main result of this research can be summarized as follows:

> *Simple dynamic load balancing algorithms using initial job placement alone can improve performance significantly, without introducing system instability.*

1.4. Organization of the Dissertation

The approach used for this research is a combination of trace-driven simulation and measurement based on an experimental implementation. Traces of job records with job arrival times and resource consumptions were collected from production a system, and used to drive a simulator that implements a number of algorithms in a loosely-coupled distributed system environment. Guided by the results of the simulation studies, an experimental load balancer was constructed in an environment of diskless workstations with file servers, and measurements were conducted using artificial workloads. The measurements verified many aspects of the simulation results, and also allowed us to study a number of issues unsuitable for simulation. With added confidence in simulation, job traces were collected from other systems, and more simulation experiments were conducted in an effort to extend the results, and to demonstrate their applicability in other computing environments.

The organization of this dissertation reflects the stages of our research effort, and the deepening and expansion of our results. In Chapter 2, we conduct a comprehensive survey of the research on load balancing done prior and concurrent to our work. Chapter 3 discusses our simulation studies. The design and implementation of our load balancer, as well as the measurement experiments performed on it, are presented in Chapter 4. Comparative studies in other computing environments are discussed in Chapter 5. Finally, the major results of our work are summarized in Chapter 6, and future directions of research are discussed.

Chapter 2

Load Balancing Research: A Survey

As an area of research, load balancing has received considerable attention since the early days of distributed systems in the 1970s. A sizable body of literature has accumulated, consisting of studies based on a number of different assumptions about the types of computing environment and workload, as well as the performance optimization objectives. In this chapter, we survey research on load balancing to date, in order to provide a comprehensive, though not all-encompassing, view of the field, and to put our work in perspective.

There are several ways to organize such a survey. To classify the various contributions, we can use the type of computing environment (e.g., a multiprocessor or a network of independent hosts), or the type(s) of hosts (homogeneous or heterogeneous, or, in queueing theory terminology, symmetric or asymmetric). A survey can also be organized according to the way a job is transferred: initial job placement versus process migration, or to the party that initiates a job transfer: source-initiative versus server-initiative. A combination of the above classifications may also be used. A close look at the literature, however, reveals an evolution in the approach and solution techniques used. The earlier work mainly studied static algorithms using mathematical programming and probabilistic methods. Analytic solutions of queueing network models and simulation were then introduced to study both static and dynamic algorithms. Trace-driven simulation technique has only recently been adopted. A number of load balancing systems have been implemented, but few measurement studies on them have been conducted.

We will follow the evolution in the approach, and use, in our survey, the last of the classification methods mentioned above. Although we are mostly interested in performance issues in load balancing, this survey is intended for load balancing in general, hence design and implementation work will also be discussed. In the following sections, each approach mentioned above will be discussed in detail, and the existing work surveyed. This is followed by a survey of the load indices that have been used.

2.1. Early Work: Static Algorithms

The algorithm adopted for load balancing is closely related to the type and amount of load and job information assumed to be known to the decision-making module(s). In static load balancing, no dynamic load information is used, and the assignments of the jobs to the processing hosts are made *a priori* using job information (e.g., arrival time, and amounts of resources needed), or probabilistically. Static balancing of workloads with user accounts has been attempted by system administrators for a long time. User accounts are assigned to the available machines in such a way that the workload generated by the users is balanced in the long run. Such a method is simple and potentially effective, but is severely limited by administrative considerations (e.g., the students in one class need to be assigned to the same machine), and often results in situations in which some of the machines are heavily congested, while others stay almost idle (refer to Table 1.1). Periodic reassignments of the user accounts may also be necessary as the user demands change.

Static load balancing may be done at a finer level, that is, at the job level. Ni and Abani proposed a *random splitting* algorithm that computes a routing probability matrix and distributes the arriving jobs among all the hosts according to the probabilities in the matrix [Ni81a]. Assuming the jobs' arrival times and their resource consumption patterns are known in advance, this algorithm attempts to minimize the average response time of all the jobs. Due to the non-deterministic nature of the algorithm, temporal congestions may develop in some hosts. Attempts to avoid such congestions lead to *cyclic splitting* algorithms that distribute incoming jobs to the available servers according to some cyclic schedule [Agrawala81] [Yum81].

Hua considered a static heuristic for job scheduling that attempts to balance the I/O loads as well as the CPU loads of the hosts, and cluster interacting processes on the same host to reduce communication overhead [Hua85]. The heuristic consists of two phases. In the first phase, an initial allocation of the jobs is determined as a starting point for the second phase, in which successive improvements are made using an iterative exchange method.

A different load balancing problem is that of *task assignment* in a distributed system. A program is viewed as a collection of modules that require different types of resources, and communicate among themselves *sequentially* - the output of one module is sent as input to another module. No two modules, however, are to execute in parallel. The computing environment is assumed to be a collection of heterogeneous processors. The execution cost of each module on each of the processors is assumed to be known, so is the communication cost between each pair of the modules if placed on two different processors. The problem is to place the modules on the processors so that the total execution and inter-module communication cost of the job is minimized.

Stone studied the case of two homogeneous processors by transforming it into a network flow maximization problem [Stone77]. An optimal module placement scheme corresponds to a minimum cutset of a network, which can be efficiently computed using the Ford-Fulkerson method or one of its derivatives. The general N processor assignment can be similarly formulated, but no efficient solution method is known.

Stone's solution of the two-processor static module assignment problem has been extended in several directions. In [Stone78], a *critical load factor* is shown to exist in a two-processor system such that, when the load factor for a module crosses its critical value due a change in the processor's load, its optimal assignment shifts from one processor to the other. The same problem with a memory constraint on one of the two processors is studied in [Rao79]. Bokhari studied the dynamic reassignment of program modules in a two-processor system based on the observation of the "phased execution" of a program [Bokhari79]. In addition to the execution and communication costs, the relocation and memory residency costs of the modules are considered. The algorithm is still static, however, because the reassignments are calculated before the execution starts. Despite some attempts to extend the results above, only the case of two processor systems has been solved so far.

The work on static load balancing generated considerable interest in load balancing research, but suffered the following three drawbacks.

1) Due to their static nature, static algorithms cannot respond to short-term fluctuations in a workload. As a result, the performance improvement potential of load balancing is not fully realized, as the loads of the hosts may be seriously unbalanced at times, although the loads over a long period is balanced.
2) They often assume too much job information to be implementable. The arrival time and execution cost of each job or module may be needed to compute the optimal placement schedule.
3) Even when the information is available, intensive computation may be involved in obtaining the optimum schedule.

These drawbacks led to the research on dynamic load balancing, in which the current system load is considered in determining job placements.

2.2. Analytic Modeling and Simulation

Because of the drawbacks of static algorithms mentioned above, much of the interest in load balancing research has shifted to dynamic algorithms that consider the current load conditions in making job transfer decisions. A large number of algorithms have been proposed, mostly heuristic in nature, as the optimum solution often requires future knowledge and is computationally intensive. Since it is impossible to survey all of them, we will discuss some of the

representative ones in the following sections.

The most widely used approach for studying dynamic algorithms is analytic modeling and simulation. For analytic modeling, the computer system is modeled as a queueing network with job arrivals and their resource consumptions following certain probabilistic patterns. *Queueing network solution techniques* are used to compute performance measures [Chow77] [Chow79] [Eager86a] [Eager86b] [Lee86] [Wang85]. Due to the limitations of the solution techniques, simulation is often resorted to for approximate solutions [Hsu83] [Livny82] [Ni85] [Stankovic84].

Livny and Melman point out that, in a system of multiple M/M/1 servers, the probability of some server being idle, while some other server has multiple jobs queued up (idle-waiting state) can be very high; hence, load balancing is likely to improve performance [Livny82]. Three practical algorithms are proposed and studied using simulation. However, the results are strongly affected by the unreasonably high communication channel utilization caused by the extremely small job sizes (the mean CPU time is 30 milliseconds). Empirical research has shown that the network load due to load balancing is not likely to saturate a local area network like an Ethernet [Lazowska86].

Eager, Lazowska and Zahorjan studied three dynamic algorithms [Eager86b]. When a job arrives at a host with a load above a given threshold, either the job is sent to a randomly-selected host, or a few of the remote hosts are probed, and the job is sent to a host among the probed ones that has a light load. The system is modeled as multiple M/M/1 servers connected by a network. Not only are the hosts assumed to be homogeneous (with the same processing rate), but the arrival rate and mean service time of the jobs at each host are also assumed to be the same. While the job transfer cost is considered (in terms of CPU time spent), the cost of exchanging load information is ignored. To solve the model, an approximate analytic method is used that is shown to be asymptotically accurate with regard to the number of hosts. The authors conclude that fairly simple algorithms can improve the average response time significantly, while more complicated ones are not likely to provide much further benefit. In another paper, the same authors compare source-initiative and server-initiative algorithms, and point out that, with initial job placement alone, the former class of algorithms offer better performance [Eager86a]. If the system is under very heavy load, and the cost of migrating an executing job is not much higher than that of transferring an arriving job, a server-initiative algorithm may perform well. A similar server-initiative algorithm without any assumptions about the network's topology is analyzed by Ni et al. [Ni85].

Since collecting load information used by a dynamic algorithm incurs overhead, the algorithm may have to use out-of-date information in job placement decisions. This may cause system instability. Algorithms trying to avoid or

reduce instability have been proposed, mainly using some randomization scheme to avoid multiple source hosts selecting the same host for job transfer. Hsu and Liu proposed algorithms in which the destination host is selected using a probability vector much in the same way as in the static, nondeterministic algorithms, except that the values in the vector are computed periodically using recent load information [Hsu86].

In studies based on the queueing network modeling approach, the workload is represented by probability distributions. The most widely used job arrival pattern is Poisson, while the amounts of resources demanded by the jobs (e.g., CPU time) are assumed to follow an exponential distribution. Due to the limitations of analytic solution techniques, the model's complexity has to be kept quite low, and approximation may be unavoidable. To study more sophisticated algorithms and/or more realistic models of the system, simulation is often resorted to. However, the workload model still remains a probabilistic one.

Measurement studies of the workloads being processed by production systems, performed by this author and others, show that many of the workload assumptions made in the past are far from reality ([Leland86], [Cabrera86], and Chapter 5). For example, the distribution of the amount of CPU time consumed by jobs and that of the number of I/O operations are usually highly skewed, with a ratio between the standard deviation and mean of between 4 and 40. Hence, the popular exponential distribution, which has a ratio value 1, is a very poor representation of CPU and I/O resource consumptions. In Chapter 5, measurement data from three different computing environments will be presented, and evidence will be given that the performance of load balancing is strongly dependent upon the job resource demand distributions. Unfortunately, to the author's knowledge, few researchers in the past have taken the time to verify that their workload model reflected, at least to some extent, reality, or to provide some argument for their assumptions --- the Poisson-exponential model has become the default workload model. Measurements also show that workload in production systems are often nonstationary, with bursty job arrivals and constantly fluctuating loads [Zhou87a]. Consequently, the steady state results of the modeling studies may not be trustworthy.

It is intuitive that realistic workload assumptions are crucial for the results to be reliable; indeed load balancing is based on exploiting the dynamics of the workload. On the other hand, simple analytic and simulation studies have shed light on a number of fundamental properties of load balancing. A promising approach, then, seems to be retaining the queueing network model for the system, but adopting more realistic workload models. Trace-driven simulation is such an approach. It should be pointed out that the difference between stochastic simulation and trace-driven simulation is in the method the workload is represented, not in the degree of system complexity that can be simulated. Indeed, a stochastic simulator can be easily modified to read in job traces instead

of generating jobs using probability distributions [Ferrari78].

2.3. Trace-driven Simulation

Instead of having to make assumptions about the workload, and trying to justify them, a job trace may be collected from a production system to drive a simulator, or to be used to produce probabilistic models. While the criticism can be made that the data collected from a particular system may be limited in its applicability to other systems, it is equally valid to say that the simple probabilistic assumptions may not represent *any* workload well. Instead of starting with assumptions unsupported by empirical data, it seems more reliable to start with measurement data from a particular system, and try to assess the generality of the results derived from them. If necessary, data from other environments should be collected, and the results compared.

Besides the advantage that any simulation has of being able to handle greater system complexity, trace-driven simulation uses measurement data to represent the workload. Few studies of load balancing using this approach have been reported in the literature. Leland and Ott conducted a trace-driven simulation study of load balancing using data collected from VAX-11 machines running Berkeley UNIX [Leland86]. Both initial job placement and process migration were studied, and significant improvements in the average job response time were observed.

2.4. Implementation

Ultimately, the purpose of all the modeling and simulation studies should be to provide guidance in the design and implementation of load balancers in computer systems. A number of load balancing implementations have been reported in the literature [Hwang82] [Hagmann86] [Bershad85] [Ezzat86]. In almost all of them, a special syntax for command submission has been introduced to inform the system that the command is eligible for load balancing. In some cases, a specially constructed version of an applications program is needed for its remote execution. The operating system had to be changed in many cases in order to make remote execution possible.

The earliest implementation known to the author was done at Purdue University in a UNIX environment [Hwang82]†. Special versions of compilers, assemblers, and text processing programs were constructed that called a system scheduling routine, *rxe*, to determine a "lightly loaded" destination host for execution. A modified form of the UNIX *load average*‡, with considerations for the heterogeneity of the machines in the network, was used as the load index.

---

† Both the AT&T Bell Laboratories and the Berkeley versions of UNIX were present in the system.

Bershad implemented a load balancer for the Berkeley UNIX 4.2 BSD operating system [Bershad85]. Like the Purdue system, only a few programs with large CPU time demands ("CPU hogs") were considered, and the program and data files had to be explicitly moved to the execution host due to the lack of a distributed file system. System servers (*daemons*, in UNIX terminology) were used to exchange and maintain load information represented by load averages, and to create remote jobs upon user requests.

The Process Server implemented at Xerox PARC was targeted for a workstation environment [Hagmann86]. A collection of personal workstations are supported by Process Servers that may be permanently dedicated compute servers or workstations donated by their owners when they are not using them. A central agent (the Controller) is used to collect load information and perform job placements for the entire system. Each command has to be modified to make its remote execution possible.

A load balancer for the NEST system of AT&T Bell Laboratories has recently been described [Ezzat86]. The load balancer is implemented on a number of workstations connected by an Ethernet-like local network. The name of a special program, *rexec*, must be used as a prefix to any command string to be executed remotely. *Rexec* obtains the hosts' loads, measured by their respective normalized response times, and transfers the command to the most lightly loaded one. Care was taken to use the code for a command and to create temporary files on the execution host (rather than on the initial host) as much as possible, in order to improve performance.

A load balancer consists of two parts: the mechanism for remote execution, and the policy module that collects load information and makes job placements. While the above systems have both parts, a number of distributed operating systems implemented remote a execution facility (either non-preemptive or preemptive or both) without specifying the system policy module. Examples are DEMOS/MP [Powell83], LOCUS [Walker83], Eden [Almes85], and the V System [Theimer85]. See [Harbus86] for a comprehensive survey of remote execution mechanisms.

2.5. Measurement

The most reliable, but also the most laborious way to study load balancing performance is to observe a load balancer in operation by measurement. This approach does not in principle suffer from the errors introduced in modeling and simulation. However, measurement requires the availability of a load balancer and a realistic workload running on it. To make the experiments repeatable, an

‡ The load average in UNIX is an estimate of the number of "active" processes in the system, averaged over a given period, e.g., 1 minute.

artificial workload, instead of a natural one, may have to be used. Moreover, the results may be biased towards the particular system and workload being observed, just like in trace-driven simulation.

Although a number of load balancers exist, very little measurement of their performance has been done. Ezzat drove the NEST system with a set of artificial workload and evaluated the performance of one algorithm, as well as the effect of creating temporary files on the execution host rather than on the originating host [Ezzat86]. The overhead of load balancing was measured by a few researchers (e.g., [Hagmann86]).

2.6. Load Index

In order for load balancing to be effective, some quantitative measure of the loads on the hosts, or a *load index*, is necessary. Intuitively, the higher the value of the index, the heavier the load on the host, hence the less desirable it is to transfer jobs to it. Although simple in concept, a good load index, one that reflects a host's load accurately, is very difficult to obtain. This is mainly due to the complexity of a computer (with multiple resources), and the drastically different resource demand patterns of the various jobs.

A wide variety of load indices have been explicitly or implicitly mentioned in the literature, mostly in descriptions of load balancing schemes. For example, in most of the studies using queueing network analysis, as well as in some other studies, the CPU queue length was used as the load index [Chow79] [Eager86a] [Eager86b] [Lee86] [Livny82] [Wang85] [Zhou86]. Some other authors used the CPU utilization [Alonso86] [Ezzat86]. Other possibilities include the normalized response time [Hwang82] (defined as the ratio between the response time of a process on a loaded machine and its response time on the same machine when it is empty†, the remaining processing time of all the jobs running on a host, the processing time accumulated by the active processes [Hac86], and the total processing time of the active processes [Leland86]. Functions of the above simple variables have also been used [Ezzat86] [Ferrari86]. In a number of studies in which reducing job response time was the objective of load balancing, the estimated response time was used as the load index [Bryant81] [Carey85]. Performance improvements were often reported using the indices discussed above. However, since no systematic and comprehensive comparisons between the indices have been made, their relative merits remain unclear. We note with regret that, in most cases, the authors did not even provide an intuitive (not to speak of a scientific) justification for their choice of the load index.

---

† The normalized response time varies with the process in a system with multiple classes of jobs, but a single-class system has usually been assumed.

The first systematic attempt to study the load indices to be used in load balancing was made in [Ferrari86]. Based on mean value analysis, a linear combination of resource queue lengths was proposed as a load index. In that linear combination, the coefficient of a resource queue length is the amount of service time that the particular job being considered requires from that resource. Thus, if an incoming job requires s_j seconds of service from resource r_j, and the queue length of resource r_j is q_j, then the load index li of the host, *as perceived by this job*, is

$$li = \sum_{j=1}^{N} s_j \times q_j$$

where N is the total number of resources in the host for which there is queueing. This index was evaluated with measurement experiments under a production time-sharing workload [Zhou87a].

Ferrari's index is response time oriented, and job dependent. Instead of a unique value at a particular moment in time, the load of a host differs for different jobs because of their varying resource demands, which are assumed to be known upon the job's arrival. This assumption enables us to predict the response time of a job more accurately, hence to make better load balancing decision. However, while we have found some simple relationships between the arguments of a job (e.g., the text file to be formatted, or the source program to be compiled) and the job's resource demands, the assumption that the demands of a job are known in advance may be too strong in many cases.

2.7. Summary

In this chapter, we surveyed research on load balancing according to the approach and solution technique used, and discussed the characteristics of the approaches. The survey shows that most of the attention has been paid to analytic modeling and simulation studies based on queueing network models, while trace-driven simulation and measurement have been largely ignored. This is unfortunate because, while the former methods provided some fundamental insights into the behavior of load balancing, they are not suitable for more quantitative studies due to their highly abstract and simplistic assumptions about the system and its workload. This observation led to the decision to use trace-driven simulation and measurement as the primary methods of research in our work. To identify those results that are not particular to a specific system, workloads from several computing environments are used in simulation, and the simulation and measurement results are compared. These studies are discussed in the following chapters.

Chapter 3

Trace-Driven Simulation

3.1. Overview

In this chapter, we study dynamic load balancing using a trace-driven simulation approach. Job traces collected from a production machine are used to drive a program that simulates a loosely-coupled distributed system consisting of a number of hosts connected by a network supporting broadcast, and supported by a distributed file system. A number of representative load balancing algorithms are defined and studied in detail. The costs of message exchange and job transfer are considered so that performance comparison between the algorithms can be made on an equal basis. Our objective, however, is not to select the best algorithm, but rather to study the characteristics of various approaches to load information exchange and job placement. We are interested in the effects of load balancing performance of such factors as the system's size (in terms of the number of hosts in the system), the level of system load (as defined in the next section), the values of the tunable parameters of the algorithms, the immobile jobs (jobs that have to be executed locally), and the costs of messages and job transfers. We also want to study the behavior of the system under load balancing, specifically, the impact of load balancing on individual hosts with varying levels of load as well as on different types of jobs (big versus small, transferred versus non-transferred), and the instability that may be caused by load balancing.

The important results from this study include the following:

- Under moderate load (e.g., average CPU utilization of 60%), a load balancing scheme using any reasonable algorithm can improve the mean of job response times by 30-60%, and make them much more predictable.
- Algorithms using periodic and non-periodic information policies provide comparable performances.
- For the periodic information policies, the centralized algorithms impose less overhead on the system than the distributed ones, hence can support larger systems.

- With initial job placements alone, server-initiative algorithms are likely to perform worse than their source-initiative counterparts.
- Greater performance improvement can be gained by increasing the system size, but the improvement levels off beyond a few tens of hosts.
- Load balancing can still be highly effective when up to half of the jobs that would otherwise be eligible for load balancing must be executed locally.
- Load balancing has a beneficial effect on the performance of *every* host, even of those originally with light loads.
- Load balancing reduces mean job response time under a wide range of message and job transfer overhead assumptions.
- Large jobs benefit substantially from load balancing, while small jobs do not suffer much as a result of the load balancing overhead. The performance gains of remotely executed jobs are slightly lower than, but comparable to, those of jobs similar in size (in terms of CPU times consumed), but executed locally.

Our study also provides insights into the choice of a load balancing algorithm under different system environments and load conditions.

In the next section, we describe the system we simulate and the structure of the model. We also discuss the load and performance indices we use. Section 3.3 describes the algorithms that we study in the simulation. The simulation results are presented and analyzed in Section 3.4. Some concluding remarks are made in Section 3.5.

3.2. Experiment Design

We first describe the job traces and the simulation model used in our experiments. The overhead costs are then discussed. Finally, we specify the load and performance metrics we use.

3.2.1. The Job Traces

A distinguishing feature of our study is the use of job traces instead of probability distributions to describe the arrival times and resource demands of the jobs†. We traced a production VAX-11/780 system running Berkeley UNIX 4.3 BSD, *Ucbarpa*, to collect job traces consisting of tuples of the format

$$<\textit{job arrival time, CPU time demand, number of disk I/O's}>.$$

---

† In a UNIX system, a *job* corresponds to a command line input by a user, and one or, occasionally, multiple *processes* are created to carry out the job. A trace consists of all the processes executed during the session. We will not insist on the distinction between job and process, i.e., we assume that each process in a trace represents a separate job.

The machine supported research and routine computing load of graduate students, staff members, and secretaries. Typical types of jobs included program compilation, text formatting, and mail handling. The machine also had a printer daemon running on it. Previous measurement studies conducted by the author [Zhou87a] show that the CPU is the most contended resource in the type of time-sharing systems from which the job traces were derived. There is usually plenty of main memory, hence little paging and almost no forced process swapping occur. The networking subsystem is not heavily loaded either. Therefore, we consider only CPU and disk I/O in our model.

Heterogeneity, either architectural or configurational, complicates the load balancing problem, and is a deviation from the primary concerns of this research. Therefore, we concentrate on homogeneous systems. In fact, to insure homogeneity and to ease the trace collection effort, sessions of job traces were collected on the *same* host for a number of days to represent a number of hosts connected by a network‡.

The selection of simulation session length is important because the boundary effects caused by jobs started before the session begins and by those finishing after the session ends may significantly affect the accuracy of the results. On the other hand, longer sessions involve greater effort in trace collection and simulation. We chose the length of each session to be four hours. Typically, about 6000 processes are created on each host during this period. Even so, some of the processes executing during a session are not included. Such processes are mostly system services that are started at system boot time and run until the system goes down, and a few very long jobs, e.g., the user command interpreter. Though small in number, they can represent a significant portion of CPU time consumption. As a result, the simulated CPU utilization during the sessions is lower than in reality, typically by 5-15 percent.

We collected the traces of 49 sessions, all of which during normal working hours, and representing moderate to heavy workload. In all the simulation experiments described in this chapter, each host in the system is fed with the trace of a *different* session. Table 3.1 shows some basic statistics about the trace data.

3.2.2. Model Structure

The simulation model is of the event-driven type [Ferrari78], and its structure is shown in Figure 3.1. We adopt a foreground-background round-robin scheduling policy for the CPU. The time quantum is 100 milliseconds, the same as that used in the Berkeley UNIX system from which the trace was derived.

---

‡ It is recognized that, by so doing, the possible temporal correlations between the loads of the various hosts are lost.

Table 3.1. Basic statistics of the Berkeley data.

Total duration:	196 hrs (49 sessions of 4 hrs each)
Total number of jobs:	297,595
Job inter-arrival time:	mean= 2.371 s, SD= 6.270 s
Job execution time:	mean= 1.492 s, SD= 19.14 s
Number of file I/Os per job:	mean=18.23, SD= 81.43
Average CPU utilization:	62.9% (from observed jobs)
Average response time (NoLB):	5.38 s
Average CPU queue length (NoLB):	2.03

When a job arrives, either from a user terminal or from another host, it is put into the foreground queue. When a job uses up its current quantum, or comes back from a disk, it is put to the tail of the foreground or background queue depending on the CPU time it has accumulated: the foreground queue if the time is less than 500 *milliseconds*, the background queue otherwise. Processes in the background queue will be scheduled for execution only if the foreground queue is empty. Since about 60-65% of the jobs have execution times below 500 milliseconds, they will stay in the foreground queue, thus receiving priority service. While the CPU scheduling policies in computer systems are usually more complicated, we feel that the above policy captures their essential features, and may be considered representative of reality. Since the level of contention at the disks is usually low under normal operating conditions in the type of system we measured [Zhou87a], we model them as infinite servers causing only processing delays, but no queuing delays. I/O operations are assumed to be evenly spread throughout the execution of each job†, and each disk I/O is assumed to take 30 milliseconds, a figure based on measurement results.

A communication network permits message passing and job transfers between the hosts. Since we are most interested in load balancing in local distributed systems, we assume that the underlying network supports broadcast (e.g., Ethernet). (This is only necessary for some of the algorithms we study.) We also assume the existence of dedicated file servers so that the costs of accessing the program and data files are roughly the same for all of the hosts. As a result, the

† Recording the times of the I/O operations during job execution would greatly complicate our trace collection effort and the model construction and simulation, without providing significant benefit, in terms of model accuracy, since the disks are not the points of contention.

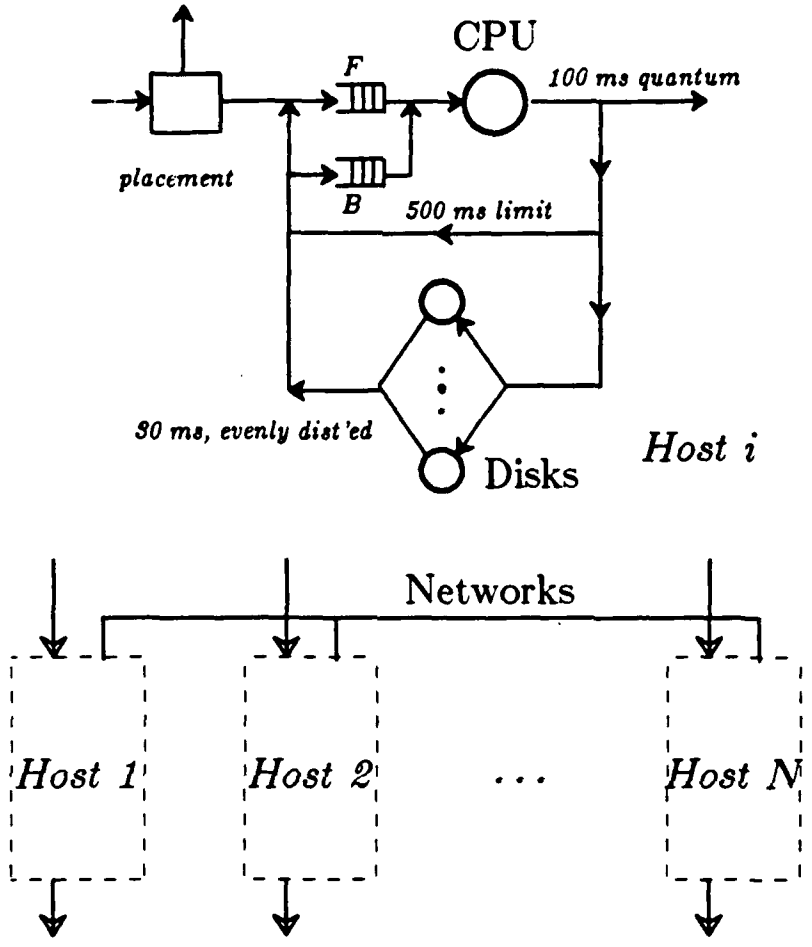

Figure 3.1. Structure of the system model used in the simulation.

files do not have to be moved with the jobs to be transferred. This assumption holds well in a diskless workstation environment, but not so well if the hosts holding the files are also used for processing. Client file caching further complicates the cost assumptions [Nelson87]. Since our trace data is derived from a time-sharing system without the support of a distributed file system, we are unable to simulate the contention at the file servers, and we also do not have measurement data on remote file accesses. The uniform cost of 30 milliseconds for an I/O operation is therefore a rough approximation.

3.2.3. Overhead Cost Assumptions

There are basically two types of overhead costs involved in dynamic load balancing. First, current loads of the hosts have to be measured and messages exchanged to make them known to the decision makers. Secondly, placement decisions need to be made and jobs transferred between the hosts. CPU time

and network bandwidth are consumed for these purposes. The second type of overhead also directly introduces extra delays in the jobs involved. (So is the first type if load information is acquired while the job to be placed is waiting, as is the case with a number of algorithms to be studied.) It has been experimentally observed that, in most current installations, local area networks, such as the Ethernet, usually have plenty of bandwidth, and the performance effects of the delay in the network are small compared to the CPU cost of executing the communication protocols [Lazowska86]. Consequently, we only consider CPU time overhead in this study. We assume that message exchange and job transfer have preemptive priority over job execution. Based on measurements from our experimental implementations of load balancing in VAX/UNIX and Sun/UNIX environments (see Chapter 4), we assume that computing the current load and sending the value to the other hosts takes 20 milliseconds of CPU time, while receiving load information and processing it takes 10 milliseconds. A job transfer is assumed to take 100 milliseconds of CPU time for both the sending and the receiving host, and to cause a 200 millisecond delay to the job being transferred. Also, we assume that the cost of a job transfer is independent of the type and size of the job because only the command line is transferred, and the program and data files are loaded from a file server.

It should be pointed out that the above cost assumptions are approximate; the actual costs in terms of the CPU times spent and the job delays introduced are sensitive to the load conditions of the hosts and of the network involved. They are also dependent on the implementation of the underlying system, as well as on the sizes of the messages. In Section 3.4.5, we study the effects of the overhead assumptions on load balancing performance by varying the message and job transfer costs.

3.2.4. Load and Performance Metrics

In order to evaluate the performances of various load balancing algorithms, we need a number of metrics. First, it is important to characterize the load on the whole system, as the performance of load balancing algorithms depends on the system's load. We choose the average CPU utilization of all the hosts over the entire session as the *load level* since it represents the level of contention for the most critical resources in the system.

We are also interested in a *load index* that we can use to measure the current load on each of the hosts, and to make job placements. Ferrari pointed out, using mean value analysis, that a linear combination of the resource queue lengths in a computer system can be an excellent predictor of job response time, with the coefficients being the estimated resource consumptions of the job [Ferrari85]. In a previous measurement study [Zhou87a], we found that a time average of the CPU queue length has a correlation of nearly 1 with the job response times of CPU-intensive jobs in a CPU-bound host, hence suggests itself as a good

load index. We use the CPU queue length, i.e., the *sum* of the foreground and background queue lengths, as our load index. In this chapter, when we say "the load of a host", we mean its current CPU queue length. More elaborate load indices based on the contentions at a number of resources, and on a smoothing algorithm, will be introduced and compared, using measurements, in the next chapter.

To measure and compare the effectiveness of load balancing algorithms, we need to define a *performance index*. We choose the mean job response time because decreasing the job response time is our primary objective of load balancing. We will also use the standard deviation (SD) of the response times of all the jobs to measure the *variability* of the job response times.

3.3. Load Balancing Algorithms

Two broad categories of load balancing algorithms are commonly recognized. In *source-initiative* algorithms, the hosts where jobs arrive take the initiative to transfer the jobs, whereas, in *server-initiative* algorithms, hosts able and willing to receive transferred jobs go out to find such jobs. Server-initiative algorithms are best supported by process migration because, in that case, jobs may be transferred upon servers' requests [Eager86a]. We studied one server-initiative algorithm based on job reservation, in addition to six source-initiative algorithms.

A load balancing algorithm consists of three components.

(1) The *information policy* specifies the amount of load and job information made available to the job placement decision maker(s), and the way by which the information is distributed.

(2) The *transfer policy* determines the eligibility of a job for load balancing based on the job and the loads of the hosts.

(3) The *placement policy* decides, for eligible jobs, the hosts to which the jobs should be transferred.

The above three component policies of a load balancing algorithm are not isolated from each other, but interact in various ways: the placement policy utilizes the load information supplied by the information policy, and acts only on the jobs determined to be eligible by the transfer policy. Because of the large number of options for each component policy, it is impossible to study all possible policy combinations in this chapter. Instead, we shall concentrate on the information policies and some of the related placement policies, while keeping the other aspects of the scheme fixed.

Seven algorithms were simulated and studied. For ease of comparison, we based the transfer policy of all the algorithms on the host load threshold T_l and the job execution time threshold T_{CPU}†. When the local load is at or below T_l,

† Notice that T_{CPU} is different from the 500 milliseconds threshold for *local* CPU scheduling.

all jobs are processed locally. Otherwise, all the jobs arriving at that host and with execution time above T_{CPU} are *eligible* for load balancing. Although job execution times are difficult to predict, it is possible to classify the jobs into two rough categories: "big" jobs which are worth considering for load balancing, and "small" jobs not to be considered. Our study of jobs submitted over 30 days show that such a classification can be made with a high success rate simply by looking at the job names. For example, a text processing job will almost certainly take over 1 second of CPU time, whereas a directory checking operation is clearly not worth considering for load balancing. Moreover, estimation errors can be easily tolerated, as long as they are not too frequent. This is supported by one of the results of this research: the performance of the load balancing algorithms is quite robust with regard to the job threshold T_{CPU} (see Section 3.4.3). In the measurements to be described in Chapter 4, this assumption about T_{CPU} will inevitably be dropped.

The following algorithms were studied:

GLOBAL

Every P seconds, one of the hosts, designated as the *load information center* (LIC), receives load updates from all the other hosts and assembles them into a *load vector*, which is then broadcast to all the other hosts. If the load of a host is the same as that sent out the last time, however, no update needs to be sent to the LIC. This applies to the next algorithm, DISTED, as well.

The placement policy of the GLOBAL algorithm, as well as that of DISTED, is as follows. The local version of the load vector is searched for a host with the lowest load, and, if this load is lower than that of the local host by Δ† or more, the job is sent to that host. If there are several hosts with the same shortest queue length, one of them is selected arbitrarily.

DISTED

Instead of reporting the local load to a centralized LIC as in GLOBAL, each host broadcasts its load periodically for the other hosts to update their locally maintained load vector.

CENTRAL

In the above two algorithms, placement decisions are made by each host using the local version of the load vector. In the CENTRAL algorithm, the LIC acts as a central scheduler for all the hosts, in addition to receiving load information from the other hosts periodically. When a host decides

---

† The optimal value for Δ depends on the workload. In our simulation, one was the best value for Δ in most cases.

that a job is eligible for load balancing, it sends a request to the LIC, together with the current value of its load. The LIC selects a host with the shortest queue length and informs the originating host to send the job there. Meanwhile, it increments its version of the destination host's load by 1. This algorithm is used in the Process Server [Hagmann86].

For the above three algorithms, it is assumed that the loads of all the hosts are known to the placement decision maker(s), with some delay. The algorithms below use less system state information, and thus have smaller overhead costs.

RANDOM

This algorithm uses only local load information. When a job is found to be eligible for load balancing, it is sent to a randomly selected host.

THRHLD

A number of randomly selected hosts, up to a limit L_p (probing limit), are polled when an eligible job arrives, and the job is transferred to the first host whose load is below load threshold T_l. If no such host is found, the job is processed locally.

LOWEST

This is similar to THRHLD except that, instead of using a threshold for the placement, a fixed number of hosts L_p are polled and the most lightly loaded host is selected. The probing stops if an empty host is found.

RESERVE

This is a server-initiative algorithm based on job reservation. Upon departure of a job, the local load is checked. If the load is below T_l, the host probes other hosts to register up to R reservations at R hosts with loads above T_l. The outstanding reservations at a host are stored in a stack so that, when a job eligible for load balancing arrives, the top reservation on the stack is used, so the job is sent to the host that made the most recent reservation. If the host's load falls below T_l, all its reservations are canceled. A modification to this basic scheme is for the sender host to probe the potential server host, and to send a job there only if the server is indeed still lightly loaded. Measurements show that this modification yields slight improvement in performance.

Leland and Ott studied an algorithm similar to DISTED using trace-driven simulation [Leland86]. The algorithms THRHLD and LOWEST above are identical to the ones studied by Eager *et al* [Eager86b]. However, we use a trace-driven simulation method to evaluate them, and we compare them to those algorithms that use a load vector. The algorithms above make placement decisions on the basis of various amounts of system state information. Since we consider the overhead costs of load balancing explicitly, a direct assessment of the appropriate amount of load information for load balancing can be made.

For comparison, we also simulated three boundary cases of load balancing:

NoLB

No load balancing is attempted; all arriving jobs are processed locally.

NoCOST

This is the unrealizable case in which the current CPU queue lengths of all the hosts are known to the transfer decision makers at no cost (in terms of CPU time *and* job delay), and the transfers of jobs are also assumed to be costless.

PartCOST

This is the partly-ideal case in which perfect load information is assumed to be known at no cost, but job transfer costs are considered.

The performance of all the algorithms can be expected to be between those of NoLB and NoCOST.

It is recognized that there exist other algorithms that can potentially produce good performance. The above seven algorithms were chosen because they represent a reasonably large collection of algorithms, and are implementable. Both server- and source-initiative algorithms are included, and different approaches to load information exchange (periodic versus on-demand) and to job placement (system-wide selection, subset, and random) are represented.

For ease of reference, we list the symbols used in this and the next chapter, together with their meanings, in Table 3.2.

Table 3.2. Symbols used in Chapters 3 and 4, and their meanings.

symbol	meaning	used in
T_l	local load threshold	simulation, measurement
T_{CPU}	job execution time threshold	simulation
P	load exchange period	simulation, measurement
L_p	host probing limit	simulation, measurement
R	job reservation limit	simulation
Δ	minimum load delta for job transfer	simulation, measurement
N	number of hosts in the system	simulation, measurement
T	averaging interval for load index	measurement
E	CPU execution time required by a job	simulation, measurement
τ	immobility factor	simulation, measurement

3.4. Simulation Results

Simulation runs with various system sizes and load levels were executed. To make the performance comparison between the algorithms meaningful, a large number of simulation runs were conducted for each system and algorithm, with different adjustable parameter values (e.g., T_l, T_{CPU}, and L_p), and the *best response time* was selected. In this way, the comparison is between the *best achievable* performances of different algorithms, and it is hoped that they reveal the quality of the algorithms. The results of the simulation experiments are presented in the following sections. We first compare the performances of the seven algorithms, then study the effects that system scale, load level, parameter values, immobile jobs, and overhead costs have on those performances, and the impact of load balancing on the performances of individual hosts and job classes. Finally, the potential problem of system instability introduced by load balancing is discussed.

3.4.1. Comparison of the Algorithms

The average response times of the seven algorithms and of the three boundary cases in systems containing 2, 4, 6, 7, 14, 21, 28, 35, 42, and 49 hosts are shown in Figure 3.2, with Figure 3.2a showing the upper range, 7-49, and Figure 3.2b the lower range, 2-7. To make the comparison between systems of different sizes meaningful, the system load levels are selected to be within a narrow range (see the load level numbers at the top of Figure 3.2), and the response times are normalized with respect to that of NoLB. Since we had a total of 49 sessions of traces, systems with smaller number of hosts (e.g., 7, 14, and 21) were simulated using several sets of traces. The results were found to be close to each other, and the average values were used in Figure 3.2.

The first observation in Figure 3.2 is that all the algorithms provide substantial performance improvements over the NoLB case (whose normalized response time is 1.0). The four best performing algorithms, GLOBAL, THRHLD, LOWEST, and CENTRAL have response times within a narrow range.

The comparison between GLOBAL and DISTED is highly instructive. Since they are the same, except for their information policies, the significant performance difference reveals the advantages of using a global agent as a relay point for load information exchange. Assume that there are N hosts in the system, and let the update period be P seconds, and the CPU time costs of sending and receiving a message plus related processing be M_{send} and M_{recv}, respectively. For GLOBAL, the overhead, in terms of the percentage of CPU time spent on load information exchanges, is

$$C_{LIC} = \frac{(N-1) \times M_{recv} + M_{send}}{P} \times 100\%$$

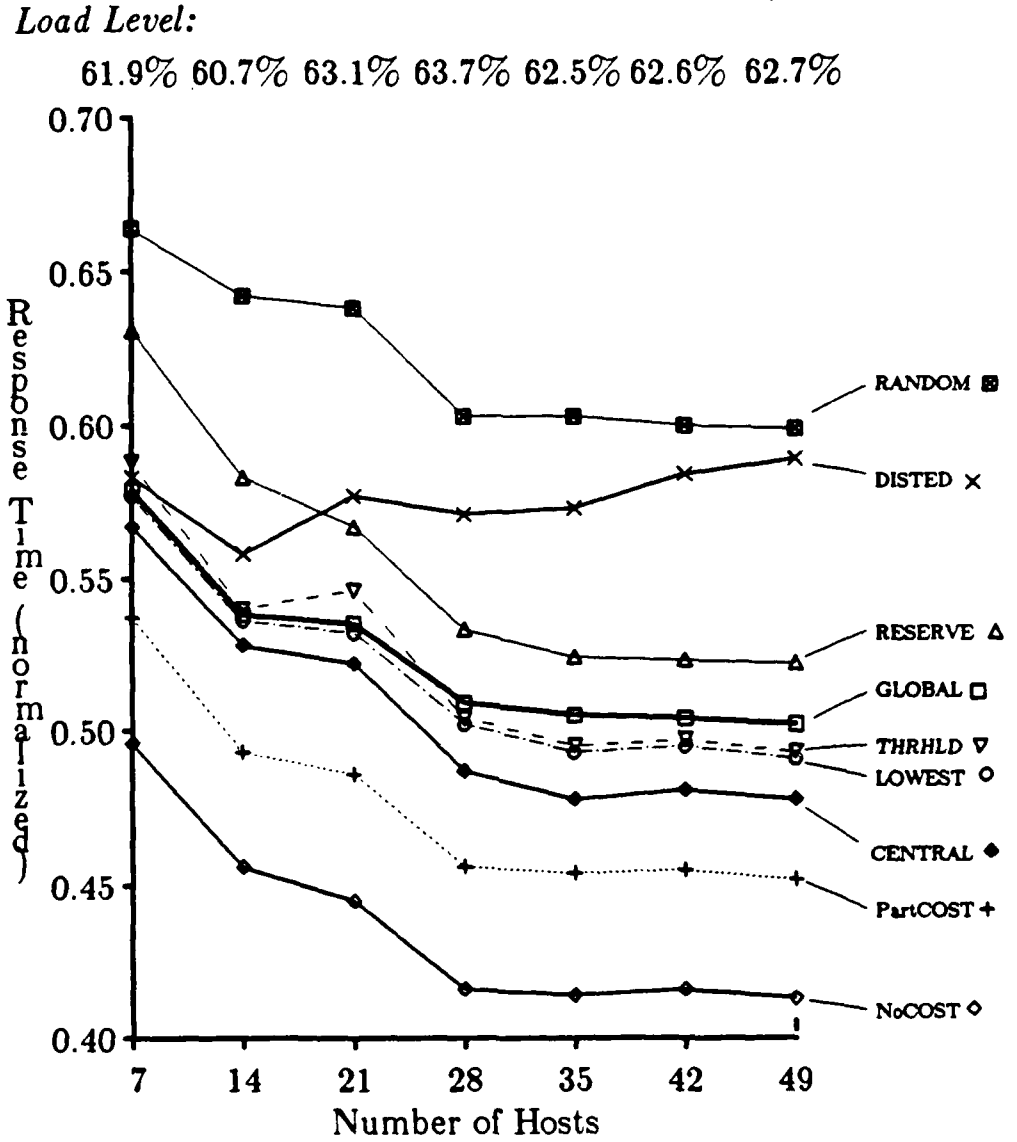

Figure 3.2a. Average response times with different system sizes (7-49 hosts; normalized against the NoLB case).

for the LIC, and

$$C = \frac{M_{send} + M_{recv}}{P} \times 100\%$$

for the other hosts. Except for the LIC, the message overhead is *independent* of the system size N. In contrast, for the DISTED algorithm, the message overhead for *every* host is

$$C = \frac{(N-1) \times M_{recv} + M_{send}}{P} \times 100\%$$

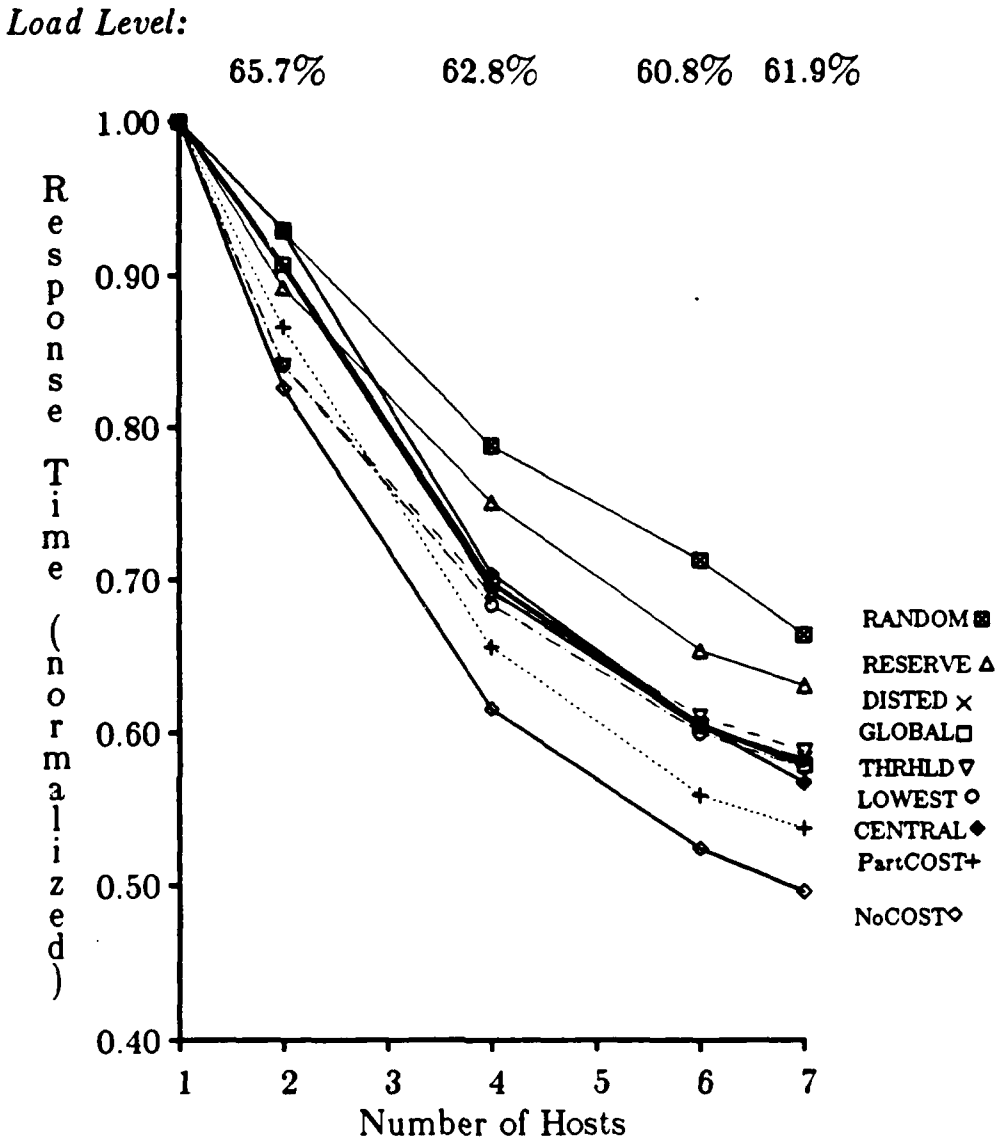

Figure 3.2b. Average response times with different system sizes (2-7 hosts; normalized against the NoLB case).

because, during each interval of duration P, every host has to process the messages broadcast by every other host. (Due to the policy of not sending out the local load information if it is the same as last time, the actual overheads of GLOBAL and DISTED are lower than presented here, typically by 40-70%, but the order analysis here is still valid.) Compared to GLOBAL, DISTED does not have a central point of failure and an extra level of indirection in the distribution of load information, but the overhead is higher for every host, and grows linearly with the system size. This is reflected in the rising curve for DISTED. Since the availability consideration is not important, for reasons to be discussed,

the GLOBAL algorithm appears more attractive than DISTED.

One somewhat surprising result from Figure 3.2 is that the two drastically different algorithms, GLOBAL and LOWEST, provide almost identical response times under a wide range of system sizes. The GLOBAL algorithm uses more extensive system state information in an effort to make optimal transfer decisions. To achieve this, load information is exchanged at a high frequency, thus incurring high overhead. In the simulation runs, the value of P that provides the best performance for GLOBAL is between 0.75 and 3 seconds. At such a high frequency, 1-3% of the CPU time in every host is spent exchanging load information. The THRHLD and LOWEST algorithms, on the other hand, do not attempt to select the globally "best" host for job transfer, but rather only select the least loaded among a small group (up to L_p) of randomly picked hosts. Although the time it takes to poll the hosts directly increases the response time of the waiting job, more up-to-date load information is used for job placement. A main reason GLOBAL is not able to perform better than LOWEST is that there exists a fundamental contradiction between the need to frequently update the load vector at each host due to the rapid fluctuations in load and the low utilization of the load vectors. If the exchange period is 1.0 second, and the rate at which transfer decisions are made by a host is one job every 10 seconds, then 90% of the load exchanges are wasted. THRHLD and LOWEST have lower overhead, do not require broadcast, and are completely distributed, thus more reliable.

It is interesting to note in Figure 3.2 that CENTRAL provides the best performance among the seven algorithms, even in a system of 49 hosts. It has been widely assumed that, in distributed systems, centralized solutions are undesirable because they tend to create performance bottlenecks and single points of failure. Such a view, however, may be too simplistic if unqualified. The best solution is environment and problem dependent. For load balancing, if the interprocessor communication is relatively efficient (such as in this chapter), and the system scale is limited (up to 50-100 hosts), the centralized approach to load information distribution and job placement may be simple and efficient. The cost of job placements is reduced for all the hosts except the LIC, as they now only need to send local load information and placement requests to the LIC, rather than maintaining system-wide load information and performing placements themselves. For the LIC, we observed that up to 35% of its CPU time may be spent for load balancing functions supporting a system of 49 hosts. (Notice, again, that a message with load information is sent to the LIC only if there has been significant change in load; consequently, the overhead is cut substantially.) Although this is a high overhead for this host, it is a small price to pay for the whole system. In return, excellent placement decisions based on up-to-date information are achieved. This explains why the performance of CENTRAL is slightly better than those of THRHLD and LOWEST, which only attempt to

select a host from a small subset of the hosts. Since a file server has frequent interactions with other hosts, it may be used as the LIC. This was done in the Process Server [Hagmann86].

For many distributed applications, availability is crucial, hence a centralized solution is not appropriate. This is not the case with load balancing, however. If the LIC goes down, some other host can quickly detect the condition and take over its role. The loss of load information is not a serious problem because load information becomes obsolete in a short while anyway. The brief interval during which load balancing is unavailable should be easily tolerable because load balancing is not an essential system service such as the naming service; its absence should in no way interfere with system operations. In fact, an implementation using essentially the CENTRAL algorithm has been reported to provide effective load balancing [Hagmann86]. In that environment, inter-host communication is extremely fast, and the global scheduler is claimed to be able to process 1000 requests per second.

The only server-initiative algorithm, RESERVE, exhibits a performance somewhat worse than its source-initiative counterpart, THRHLD. This is partly due to the restriction of initial job placement. When a host's load becomes low, it can only make *reservations* at hosts with higher loads. A reserved job takes an unknown amount of time to arrive, and, by that time, the server host may be already heavily loaded. Because of this low realization rate of the reservations, the number of reservations allowed, R, is an important parameter. Allowing only one reservation yields performance significantly worse than that shown in Figure 3.2, which corresponds to R equal to 5 (see Section 3.4.3 for more detail).

Scalability is an important issue in load balancing. In Figure 3.2, the negative slopes of all the algorithms except DISTED suggest the presence of economies of scale. As the number of hosts in the system increases, the probability of finding a lightly loaded host increases, and the average response time can be expected to decrease. This is most obvious for the NoCOST case, where the overhead costs are not considered. For the realizable algorithms, the overhead may increase with the system size, making the increase in system size a mixed blessing. Therefore, the scalability of an algorithm is an important property. It is not surprising that the scalability of RANDOM, THRHLD, LOWEST, and RESERVE is very good. (Their curves are almost parallel to that of NoCOST.) This is because the number of hosts polled by the algorithms for job placement or reservation is independent of system size. The scalability of the centralized algorithms, GLOBAL and CENTRAL, however, is also very good, at least up to 49 hosts. In contrast, DISTED scales quite badly. We can see two conflicting factors in action. On the one hand, an increase in system size makes it easier to find a lightly loaded host. On the other hand, the message overhead per host grows linearly with system size. The composite effect is a moderately rising curve for the normalized response time.

It is interesting to observe that, as the number of hosts increases beyond 28, the normalized mean response time improves very little. Therefore, an algorithm with a scalability of up to a few tens, or at most a few hundreds of hosts, seems sufficient. Beyond that point, it makes more sense to implement load balancing in clusters and perform inter-cluster load balancing using longer-term load information. This observation further enhances the value of algorithms such as CENTRAL and GLOBAL.

3.4.2. Effect of System Load

Figure 3.3 shows the average response times of a system of 28 hosts with some of the load balancing algorithms described in the last section, and under varying load levels. Since job traces are used to drive the model, we cannot control the load level of the system. However, it is essential to observe the performance of the algorithms under various load conditions. We achieved this by multiplying the job interarrival times by a factor. Thus, we were able to generate a number of points for each algorithm. Although the job stream was altered, the job characteristics (i.e., execution time, number of I/O) remained the same. We feel that such a modification to the job stream, *used within a limited range*, is unlikely to introduce significant distortions in the results†.

We observe in Figure 3.3 that, while load balancing improves the average response time throughout the range of system load levels, such improvement increases with the system's load. With a load level of 70% or over, some of the hosts are heavily congested, and the job response times without load balancing increase sharply. When load balancing is turned on, however, the whole system handles the load smoothly, causing only a moderate rise in the job response times. This characteristic is highly desirable because load balancing is most needed when the system's load is heavy. In Figure 3.3, the relative rankings of the algorithms remain the same throughout the loading range. The most promising algorithms, GLOBAL, THRHLD, LOWEST, and CENTRAL are very close to each other as in Figure 3.2.

3.4.3. Parameter Sensitivity and Adaptive Load Balancing

Once the load balancing algorithm is selected, the performance of a load balancer is still sensitive to the specific parameter values adopted. In this section, we assess the degree of such sensitivity. The adjustable parameters depend on the algorithm. For all of the algorithms, we have a local load threshold T_l

---

† Two other choices were to multiply the job execution times by a factor, and to use different job streams. However, they both would have altered the job characteristics and introduced more changes to the workload than the method we used, thus making the comparison of performances under different workload levels less meaningful.

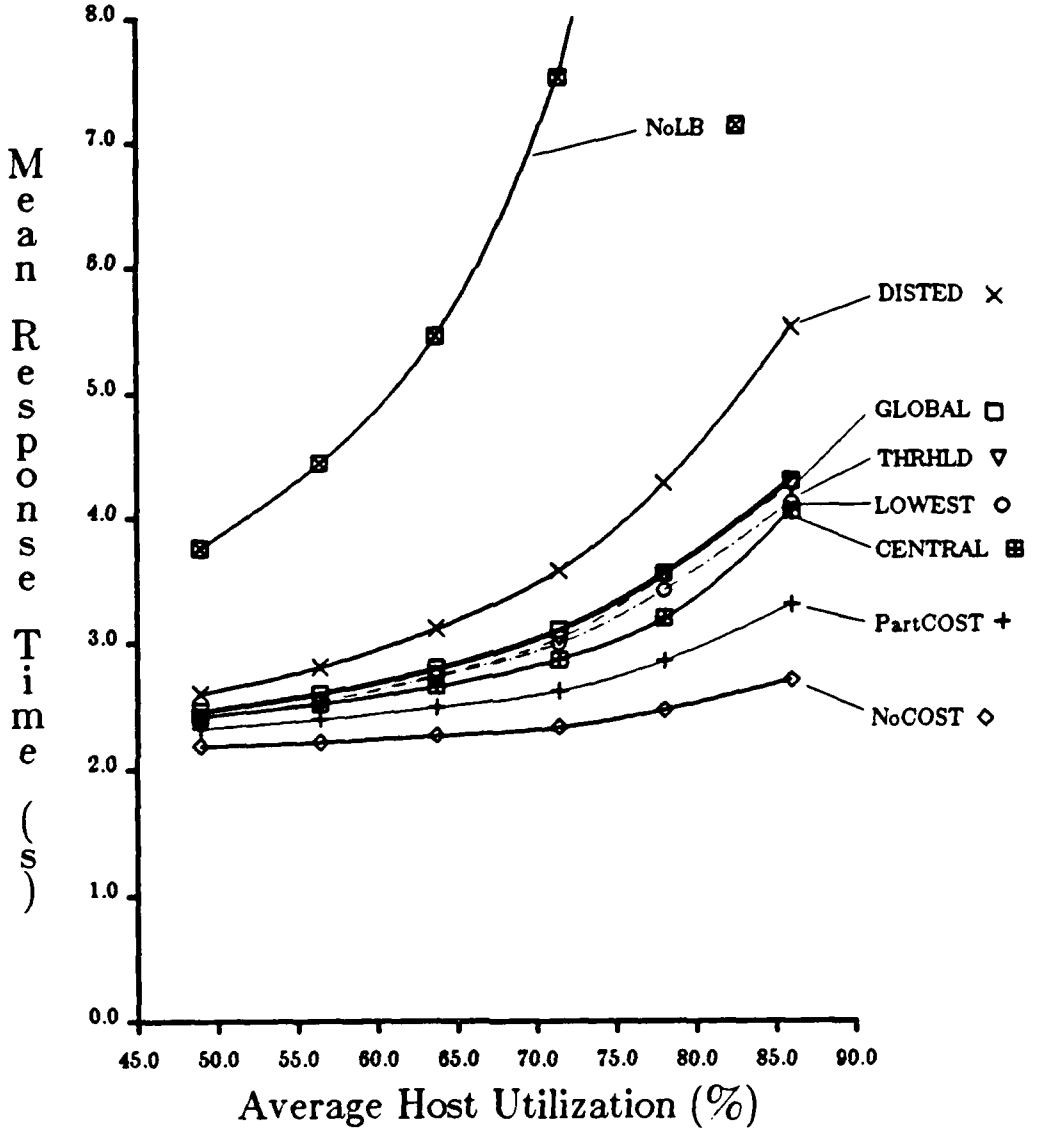

Figure 3.3. Average response times under different load levels (28 Hosts).

and a job threshold T_{CPU}. In addition, for the periodic information policies, we have the load exchange period P, whereas, for the non-periodic policies, we have the probe limit L_p. Since it is impractical to explore the effects of all the parameters for all algorithms and systems, we present only a few of them here.

Figure 3.4 shows the performance of the GLOBAL algorithm under various values of P and T_{CPU}, with T_l fixed at 0. We can see that P has a strong influence on performance. When P is too short (e.g., 0.35 second), the overhead is so high that, even though the load information on which the transfer decisions are based is very up to date, the performance suffers. On the other hand, if P is

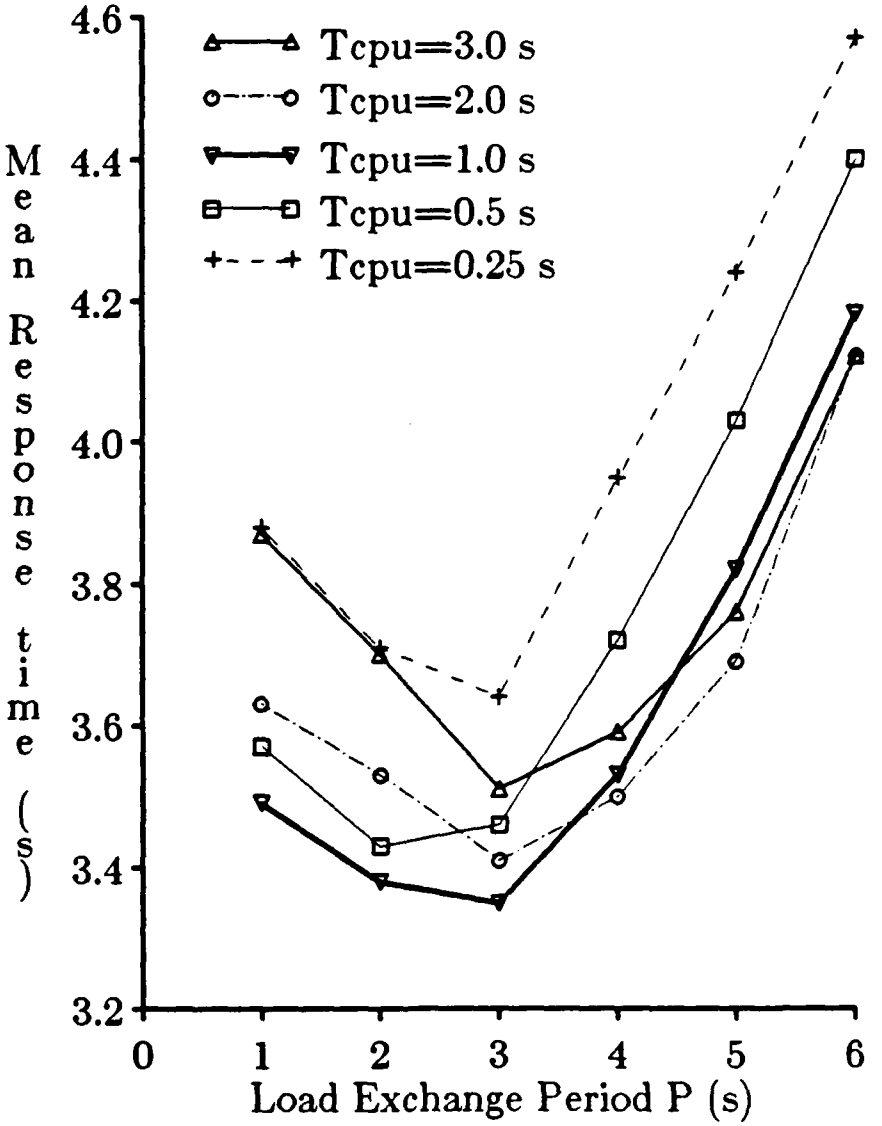

Figure 3.4. Effect of adjustable parameters on load balancing performance (GLOBAL, 14 hosts; NoLB: average response time: 7.46s, utilization: 72.3%).

too long (e.g., 10 seconds), the load information is so out of date that frequent mistakes are made in job placements, that is, jobs are often sent to hosts with equal or higher load than the local host.

In contrast, performance seems to be quite insensitive to the value of T_{CPU}: the average response time with T_{CPU} being 1.0 second is close to those with T_{CPU} being 0.5 or 2.0 seconds (Figure 3.4). This observation supports our earlier claim that only an approximate separation between large and small jobs is necessary to achieve good performance. Looking more closely at the job threshold, we again

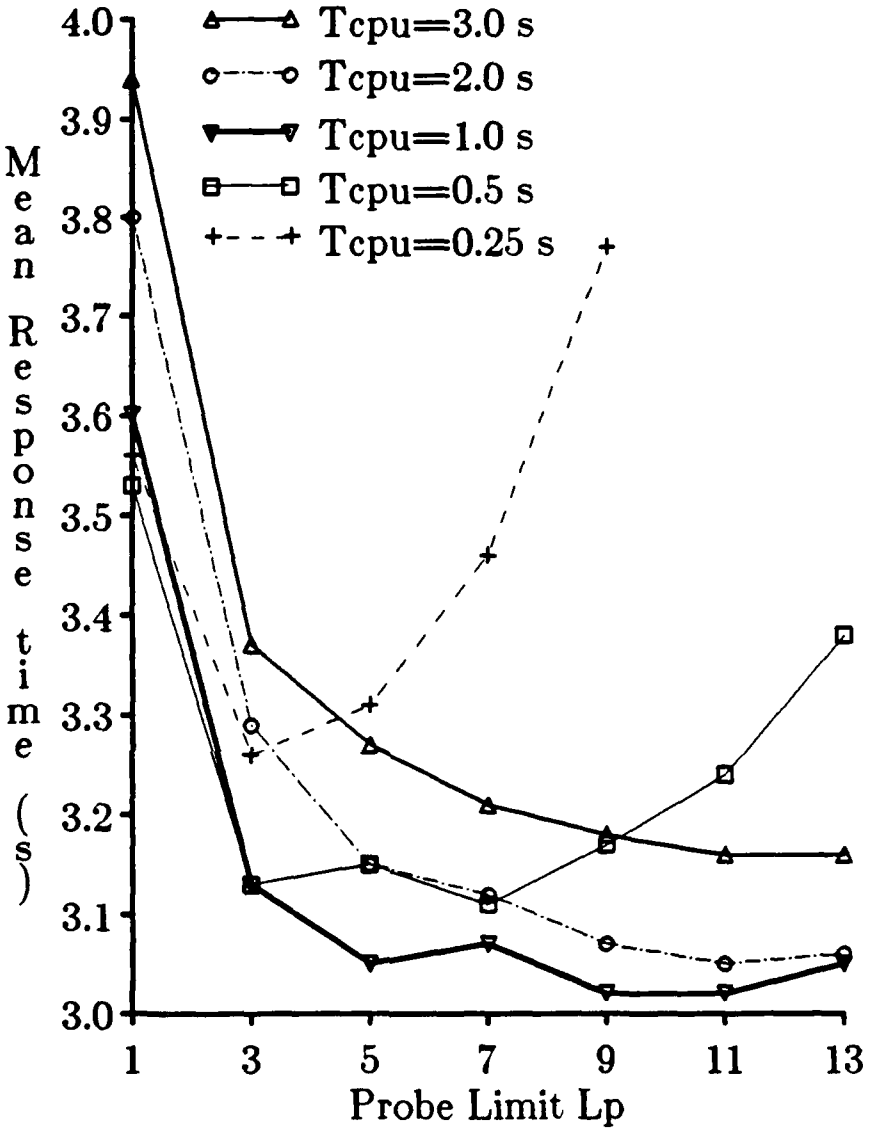

Figure 3.5. Effect of adjustable parameters on load balancing performance (LOWEST, 28 hosts; NoLB: average response time: 8.15s, utilization: 72.6%).

observe a pattern similar to that of load threshold T_l: when T_{CPU} is too high (e.g., 3 seconds), the full potential of load balancing is not realized, whereas, when it is too low (e.g., 0.25 seconds), the overhead of job transfers outweighs the benefit, and performance becomes worse. There is also an interaction between the two parameters; when the P is lengthened, the corresponding optimal value of T_{CPU} increases. Figures 3.5 and 3.5a provide similar information for LOWEST and RESERVE. As we pointed out in Section 3.4.1, allowing only one reservation in RESERVE would yield performance significantly worse than that with multiple reservations.

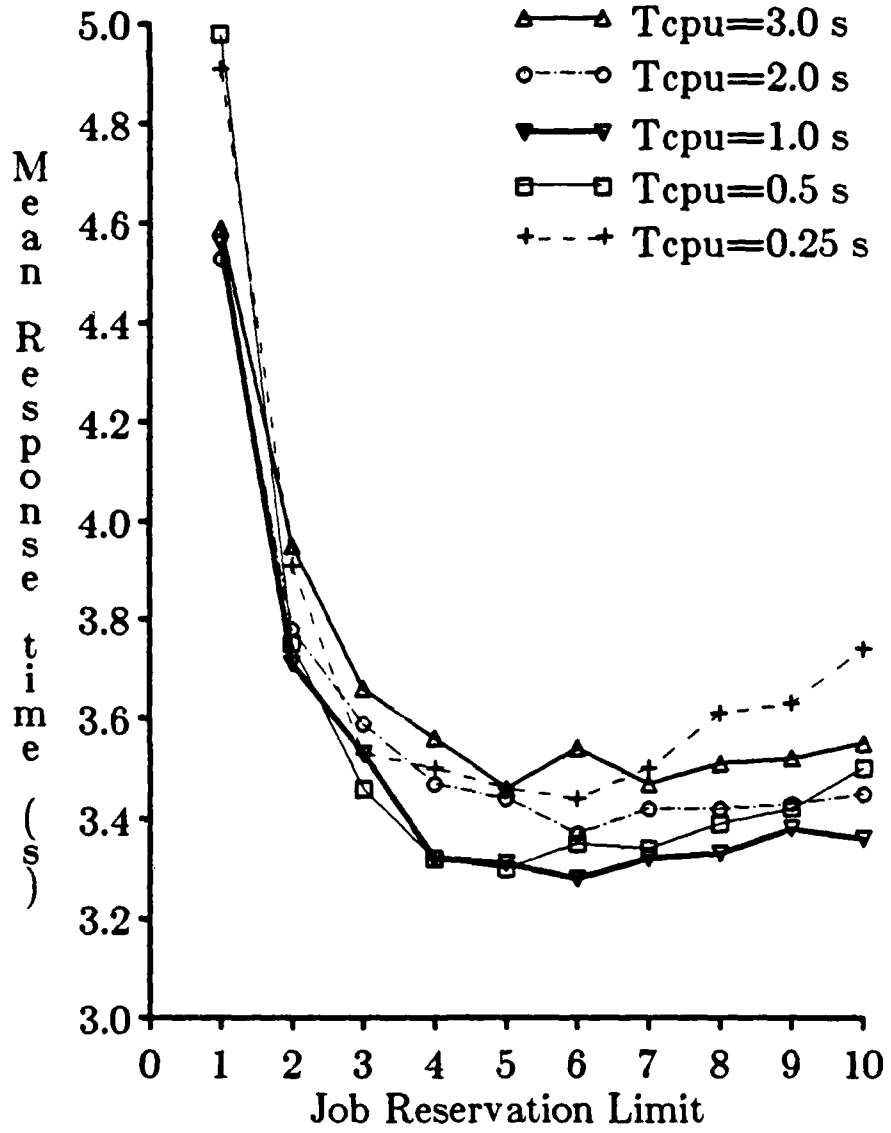

Figure 3.5a Effect of adjustable parameters on load balancing performance (RESERVE, 28 hosts; NoLB: average response time: 8.15s, utilization: 72.6%).

It is important to recognize that the combination of parameters that yields the best performance is highly dependent on the system load level. Table 3.3 attempts to illustrate this. Generally speaking, the higher the load, the higher the job threshold and the longer the exchange period should be. For LOWEST, an increase in the probing limit may yield poorer performance when the load is high.

The sensitivity of load balancing performance to the values of the parameters of load balancing algorithms suggests that some form of *adaptive load balancing* may be able to provide good performance when system load changes

Table 3.3. Optimal parameter Values under different system load levels (28 hosts).

Load Level (%)	48.1	56.2	63.3	72.6	79.1	85.1
GLOBAL						
P	0.5	0.5	0.75	1.0	1.5	2.0
T_{CPU}	0.4	0.5	0.5	0.75	1.0	1.5
DISTED						
P	1.5	2.0	2.5	3.0	4.0	5.0
T_{CPU}	0.4	0.5	0.75	1.0	1.5	2.0
LOWEST						
L_p	13	11	9	9	5	3
T_{CPU}	0.4	0.5	1.0	1.0	1.5	2.5

(The numbers are approximate, as only a sparsely allocated set of operating points in the multi-dimensional parameter space were tested for each algorithm.)

widely. Under adaptive load balancing, the system load is constantly monitored, and changes in algorithms and/or adjustable parameters are made automatically as the load changes so that the system is always operating at, or close to, its optimal point. Supporting multiple algorithms may involve complicated implementation. Furthermore, for the most promising algorithms, GLOBAL, CENTRAL, and LOWEST, the performance differentials are quite small. Consequently, the gain from switching algorithms is probably insignificant, and therefore not worth the effort. However, this is not, and cannot be, a general statement: for environments different from ours, and for algorithms other than the ones we studied, using different algorithms under different loads might turn out to be quite advantageous.

In contrast to algorithmic change, parameter adjustments are much simpler, and capable of significantly improving performance when the system load fluctuates widely. Here, we need a system-wide mechanism that monitors load conditions and makes adjustment decisions. GLOBAL and CENTRAL are the most appropriate algorithms for this purpose. The LIC periodically receives load information from all the hosts, and can use such information to deduce the system state. It can then send to the hosts the parameter values they should use. In a heterogeneous system, the values could conceivably differ from host to host.

3.4.4. Effect of Immobile Jobs

Throughout our studies so far, we have assumed that the jobs are *mobile*, that is, they can be executed on any host in the system with exactly the same results. Although this assumption holds for a large subset of the jobs, we do observe that some of the jobs are in reality immobile. Examples include jobs that perform local services and/or require local resources, such as system daemons, login processes, mail and message handling programs, and so on. There are also highly interactive jobs, such as command interpreters and editors, for which remote execution is likely to result in poor performance due to network latencies. Any implementation of load balancing must take these immobile jobs into consideration. We define the *immobility factor* to be the percentage of the eligible jobs that *have to* be executed locally†. By varying the value of the immobility factor, the effect of immobile jobs may be studied. For a system of 28 hosts with an average CPU utilization of 63.7%, the results are shown in Figure 3.6.

The concave shapes of the curves are encouraging, as they indicate that effective load balancing is still possible even if a significant proportion of the eligible jobs are immobile. For an immobility factor of 0.6, the mean response time is only slightly higher than that for the case in which all jobs are mobile (i.e., when the immobility factor is 0). This observation is not surprising because load balancing is achieved by transferring only a fraction of the eligible jobs anyway. (Typically, 50-70% of the eligible jobs, or about 10-20% of all the jobs, were actually transferred in the simulation experiments.) Consequently, even though many of the eligible jobs are immobile, the rest of them can still produce most of the performance benefits due to the balancing effect.

3.4.5. Effect of Overhead Costs

Throughout this chapter, we used the overhead assumptions discussed in Section 3.2.3 for the costs of sending/receiving messages, and transferring jobs. These assumptions are based on measurements performed in the systems we simulated, hence are reasonable reflections of reality. On the other hand, it is important to assess the *sensitivity* of the results in this chapter to these overhead assumptions. By doing this, we can have a better understanding about how well the results would apply in environments with different overhead costs (e.g., the environment at the Lawrence Berkeley Laboratory to be discussed in Chapter 5).

---

† In our simulator, we mark the eligible jobs mobile/immobile when they arrive, using the immobility factor as the corresponding probability. Hence, statistically, the average execution times of the mobile and immobile jobs are the same, as we assume a single class of jobs.

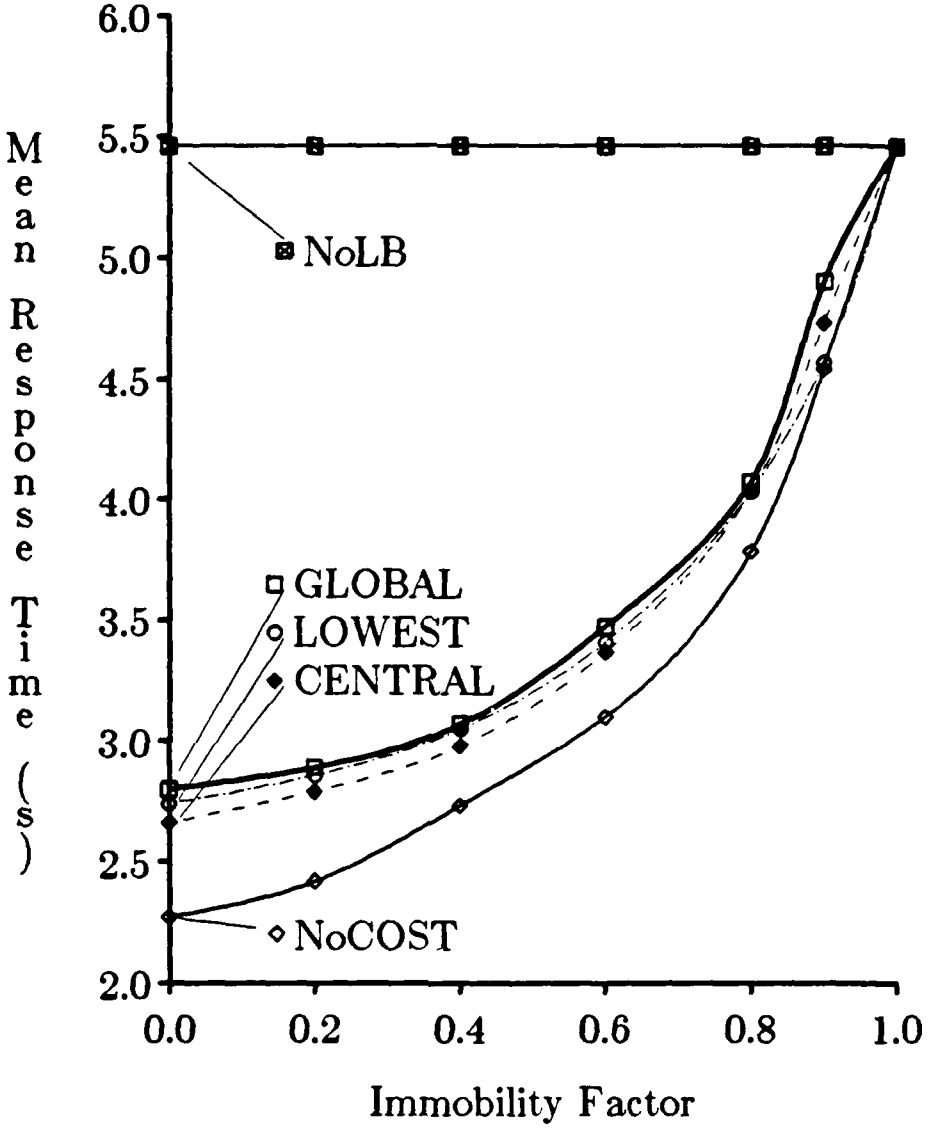

Figure 3.6. Effect of immobile jobs (T_{CPU}=1.0 s).

We vary the overhead assumptions along two dimensions: the cost of sending/receiving a message and of the associated processing, and the cost of transferring a job. The normalized mean response times of all the jobs are plotted in Figures 3.7 and 3.8 along these two dimensions. All the response times are the best achievable ones, as described at the beginning of Section 3.4.

Since RANDOM, PartCOST, and NoCOST do not exchange load information explicitly, their corresponding curves are flat; all the curves for the other algorithms, however, show a positive slope (Figure 3.7). This agrees well with our intuition that performance would degrade as message overhead increases.

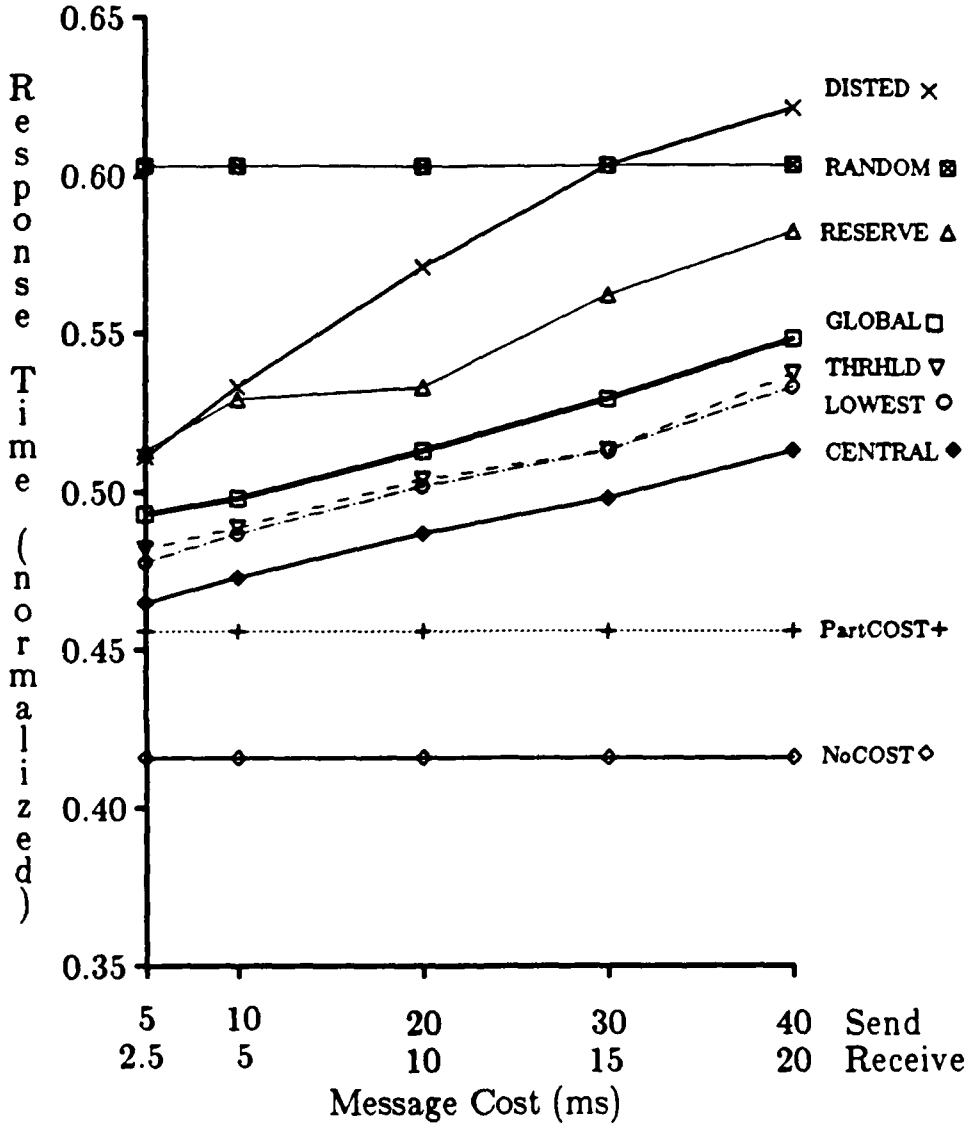

Figure 3.7. Effect of message exchange costs on mean response time.

On the other hand, it is remarkable that, even when sending a message costs 40 ms and receiving a message costs 20 ms, the performances of the four best-performing algorithms, GLOBAL, THRHLD, LOWEST, and CENTRAL, do not suffer much degradation; note also that their curves are almost in parallel to each other. The curve for DISTED, in contrast, exhibits a steeper slope, reflecting the greater dependency of DISTED upon the message exchanges. Throughout the range of message overhead considered, the relative rankings of the algorithms, except DISTED and RANDOM, stayed the same, and are similar to those shown in Figure 3.2.

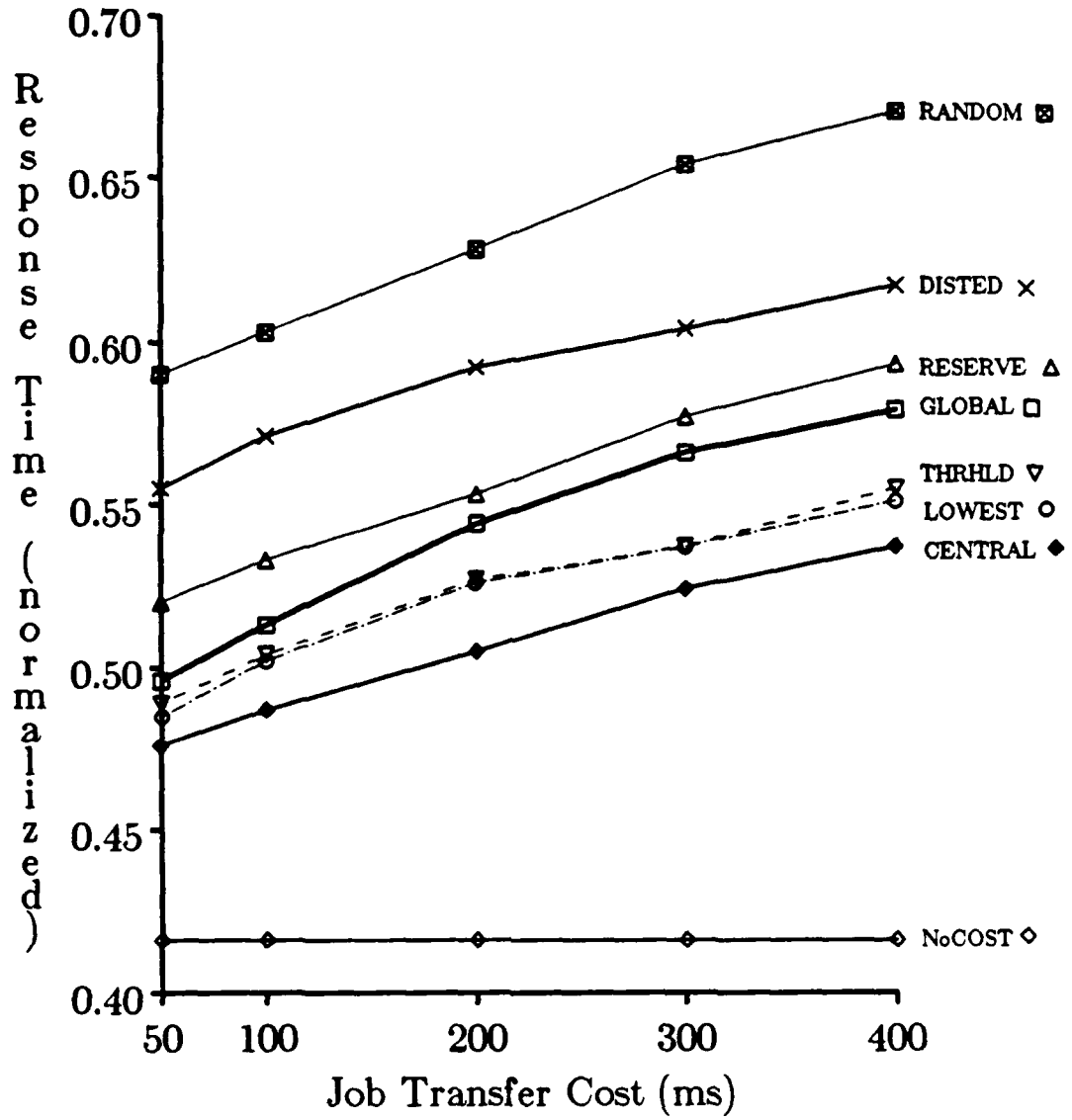

Figure 3.8. Effect of job transfer costs on mean response time.

The job transfer cost used in Figure 3.8 refers to the CPU processing time for each of the sending and receiving hosts. The total delay to the transferred job is assumed to be twice as much. We can draw observations from Figure 3.8 similar to those from Figure 3.7, except that all the realizable algorithms degrade at about the same rate now, as all of them have to incur job transfer costs.

From the above discussion, we conclude that load balancing performance is quite insensitive to the overhead assumptions, as long as the overheads are not too high. Our comparisons of the performances of the algorithms remain valid

under quite different overhead assumptions. It should be pointed out that the reasonably good performances of the algorithms under high overhead assumptions were achieved by adjusting the values of the parameters. Typically, the load exchange period was lengthened, the load and job thresholds increased, and the number of hosts to be probed reduced.

3.4.6. Impact on Individual Hosts

In the above sections, we compared the performance of a number of algorithms, and studied the effects that system scale, load level, parameter values, and immobile jobs have on load balancing. We now study the impact of load balancing on individual hosts and job classes. We fix the system under study to be one with 14 hosts and CPU utilizations as shown in Figure 3.9.

Previous studies of load balancing have frequently assumed that the hosts in the system are subjected to the same level of load [Eager86a] [Eager86b] [Livny82] [Wang85]. (E.g., the job arrival rates and the processing rates of all the hosts were assumed to be the same.) However, this is usually not the case in production environments. It is very interesting to study the effect of load balancing on the individual hosts, especially those originally with light loads. At the beginning of this research, we conjectured that, while load balancing may *improve both the performance of the system and that of the heavily loaded* hosts, the lightly loaded ones would suffer degradation in their performance because additional jobs are transferred to them. We were, therefore, pleasantly surprised by the simulation results. The average response times of the individual hosts, with and without load balancing, are shown in Figure 3.10†. (Note that the averages in Figures 3.10 and 3.11 are for jobs that *originated* at a particular host, rather than for those that were *executed* at the host. In other words, they are the user-perceived mean response times.) As can be observed, the performances of *all* hosts generally improved, with the hosts under heavy loads showing greater improvements. Figure 3.10 clearly demonstrates the power of dynamic load balancing: system performance may be greatly improved by taking advantage of the temporal differences among the hosts' loadings, and even hosts with lighter loads may benefit as congestions on them, though infrequent, can be relieved by other hosts. It should be pointed out that the above observation is valid only within a limited range of host CPU utilization. The 14 hosts used in Figure 3.10 have CPU utilizations between 50% and 75%. Some other simulation experiments we ran showed the mean response time of a very lightly loaded host increasing moderately under load balancing. This is also the case for some of the measurements presented in the next chapter.

---

† Host 1 is used as the LIC for all the experiments with GLOBAL described in this chapter.

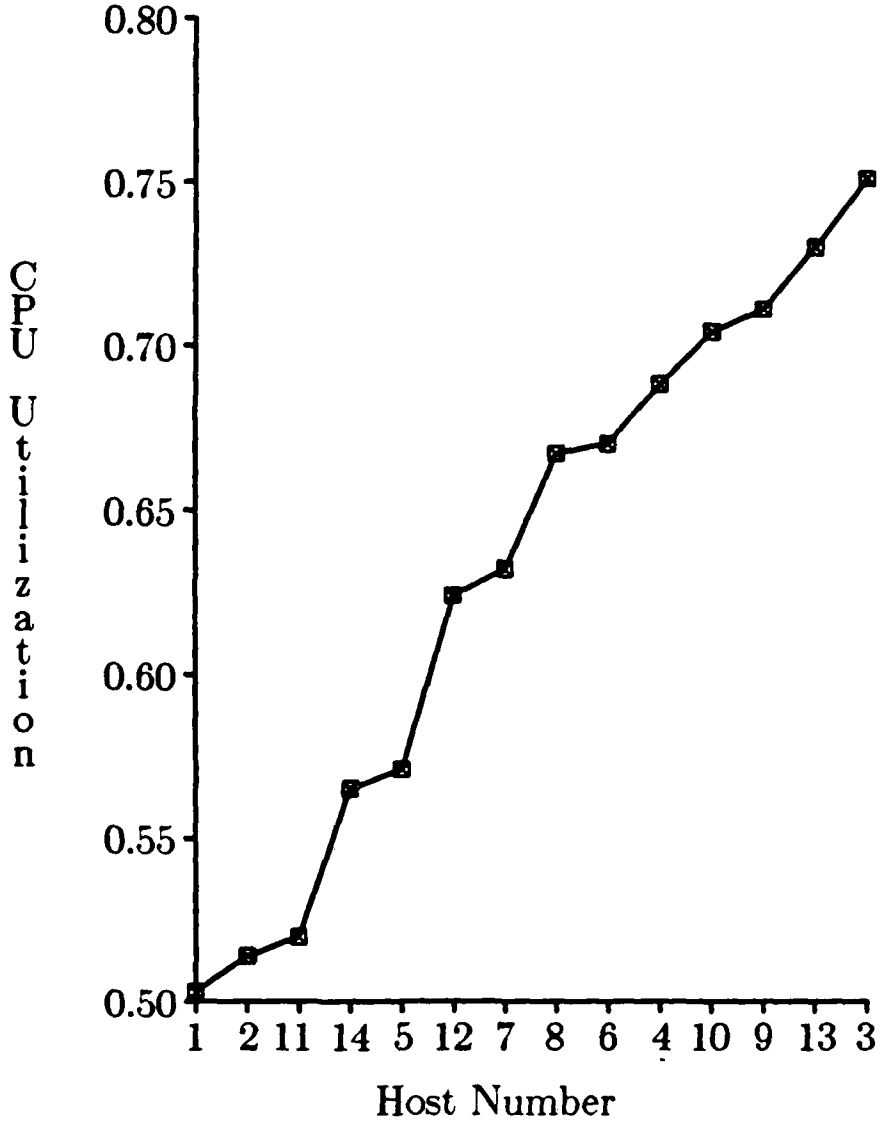

Figure 3.9. Host CPU utilizations of a 14 host system
(sorted by host CPU utilization; average utilization with NoLB: 63.3%).

Another beneficial effect of load balancing is that it makes the response times more *predictable*. In many environments, this is even more important than the reduction in the mean response time. Figure 3.11 provides a direct measure of this effect: while the average response time is cut by a factor of 1.5 to 2, its standard deviation is cut by a factor of 2 to 4. The values in Figures 3.10 and 3.11 were observed when the system was moderately loaded (the utilization for the NoLB case is 63.3%), and with moderate imbalances in host loads. The improvements in the mean and standard deviation of response time were found to be more drastic when the system load level was higher, and/or the host loads

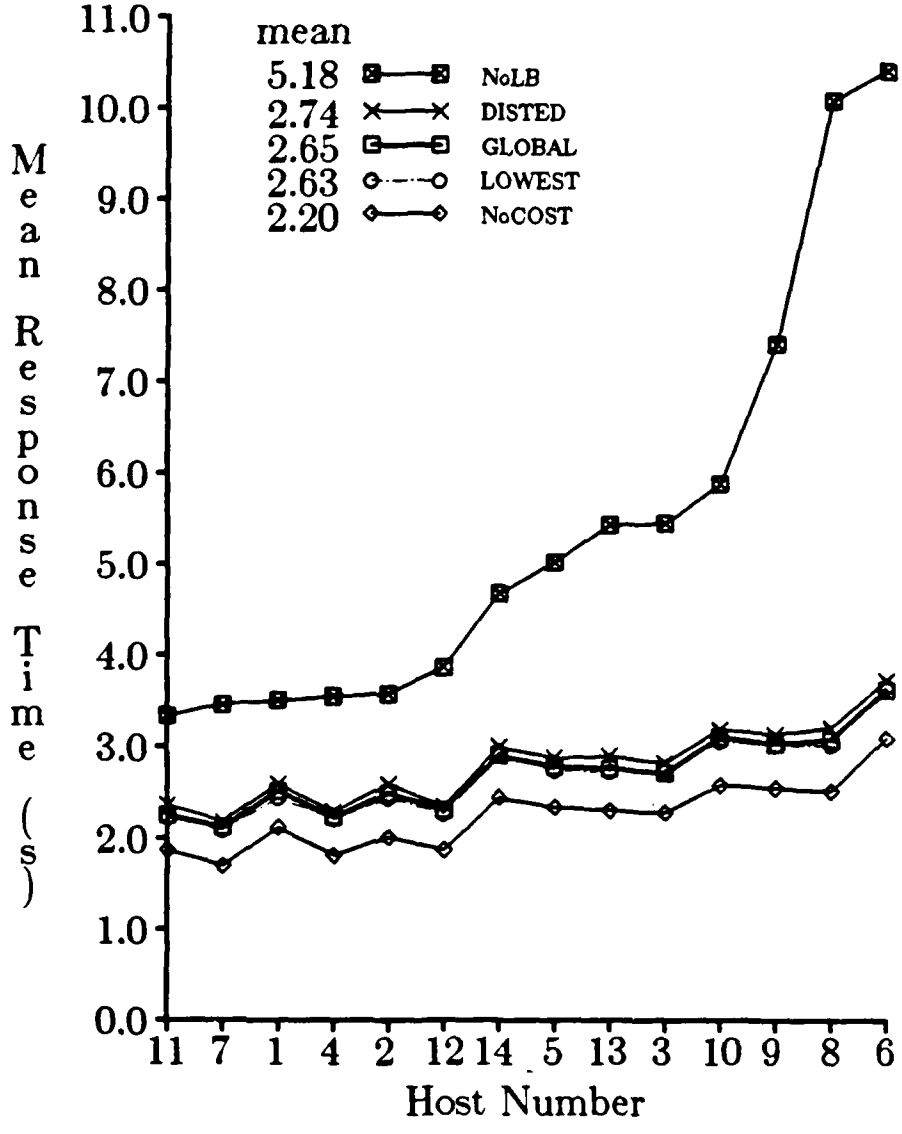

Figure 3.10. Mean response times of individual hosts (utilization with NoLB: 63.3%).

more imbalanced (see Sections 3.4.2, 4.5.1, and 4.6.1).

The term load balancing has in it the implicit meaning of equalizing the loads of the participating hosts. Though this is not our direct objective, the equalizing effect of the algorithms studied in this chapter can be clearly seen in Figure 3.12. This was observed to be more pronounced with GLOBAL and DISTED than with THRHLD and LOWEST because of the attempt of the former two algorithms at system-wide optimization.

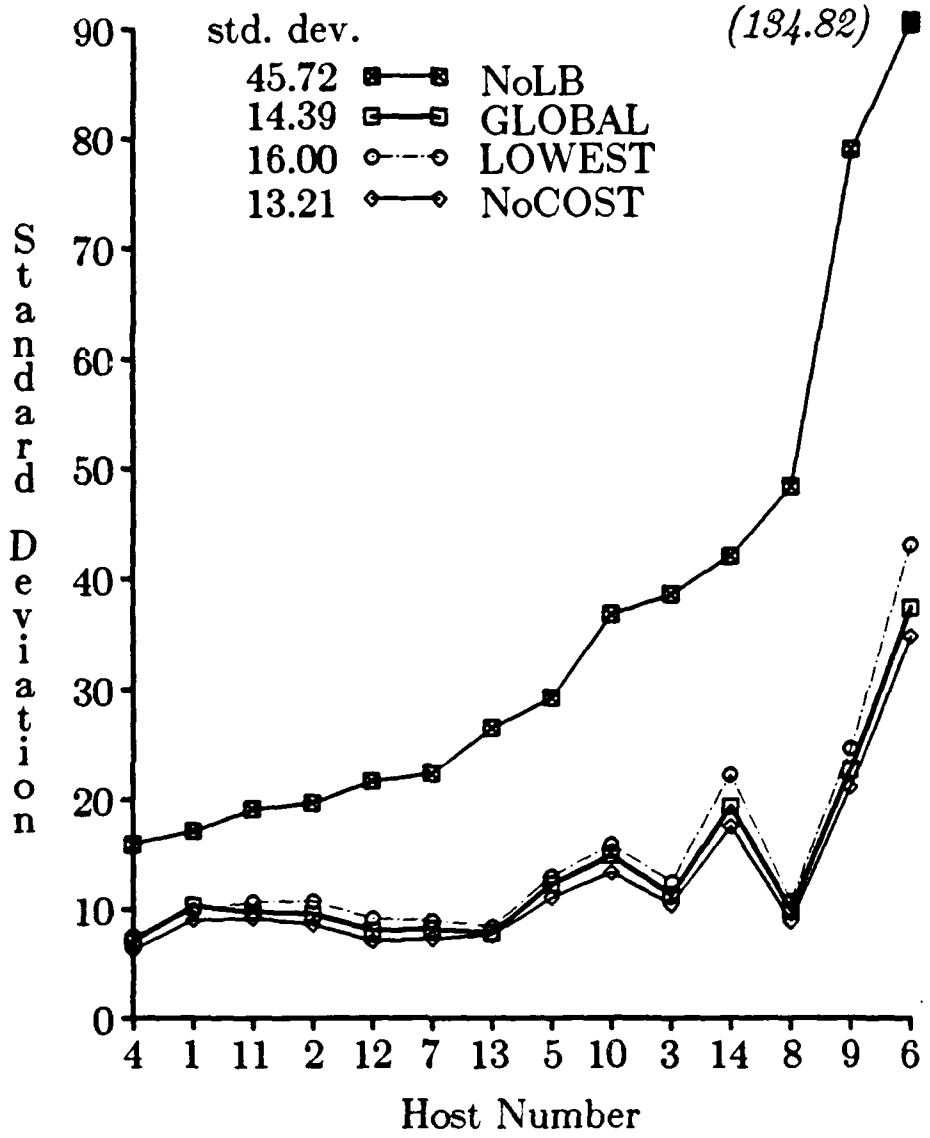

Figure 3.11. Standard deviation of response times of individual hosts (utilization with NoLB: 63.3%).

While balancing the hosts' loads over long periods (half an hour or over) is definitely desirable, it is perhaps more important to balance the loads over short periods, such as one minute. We sampled the CPU queue length of each host every second in this 14-host system, and computed the 30 second averages. Figure 3.13 shows the load index values of the most and least loaded hosts, with and without load balancing, over time. With NoLB, the hosts' loads fluctuate widely. With GLOBAL, however, the curves of all the hosts become very similar to each other. Although there still exist fluctuations, they are at a lower level, and of much smaller magnitudes. It should be pointed out that the ability of

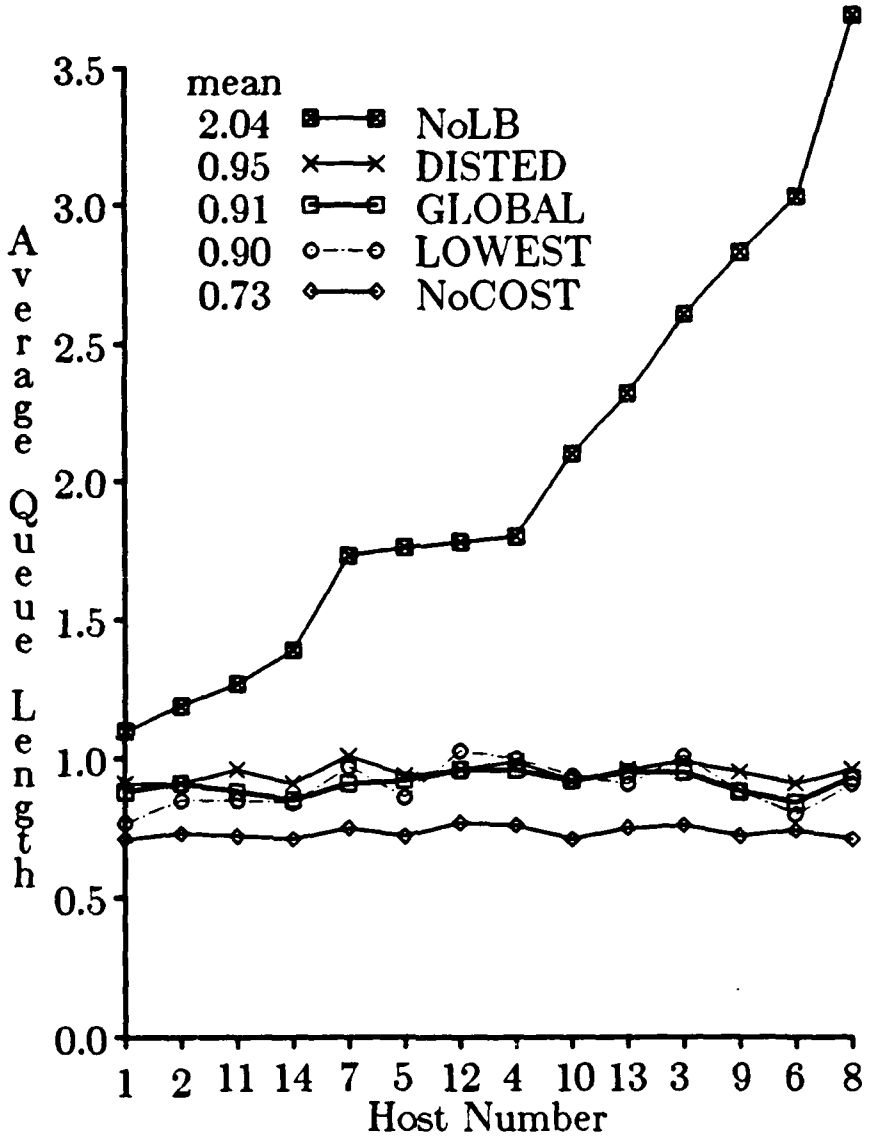

Figure 3.12. Average queue lengths of individual hosts under different load balancing algorithms (utilization without load balancing: 63.3%).

load balancing to reduce temporal load fluctuations heavily depends on the workload. The trace data we used are characterized by a large number of small jobs (about 1500 processes/hour/host, with average execution time of about 1.5 seconds). Hence, the smoothing effect is strong. In contrast, in the measurement study of load balancing to be presented in the next chapter, we used a workload with large jobs, and the smoothing effect was found to be limited.

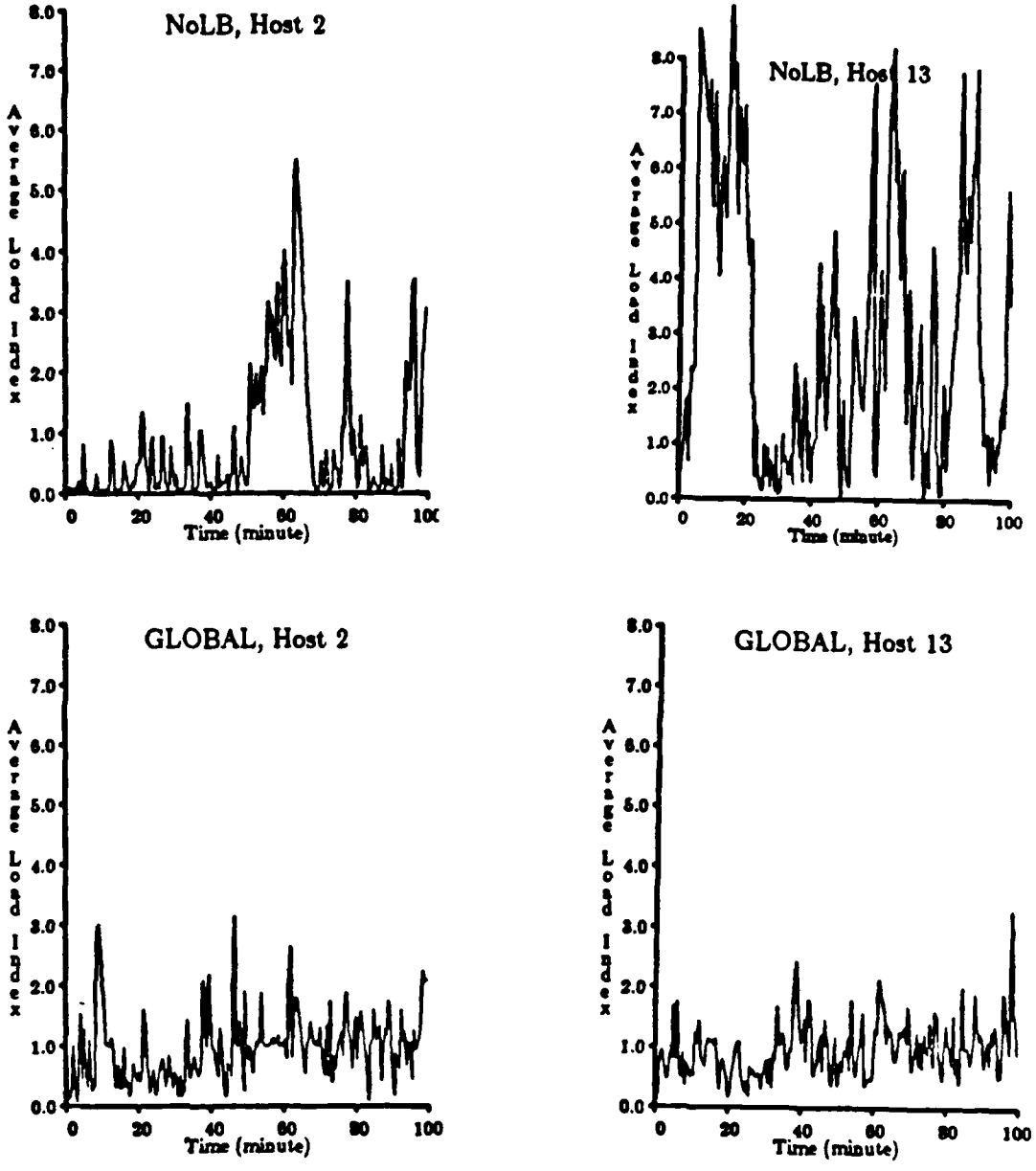

Figure 3.13. The loads on sample hosts as a function of time.

3.4.7. Impact on Each Job Class

It may be expected that load balancing has varying influences on jobs of different classes and sizes. We divide all the jobs into two large categories: BIG---those with execution times above the job threshold, T_{CPU}, and SMALL---those with execution times at or below T_{CPU}. For BIG jobs, we further divide them into Remote---those transferred to other hosts for execution, and Local---those

executed locally. Figure 3.14 shows the average response times of each class of jobs, with NoLB and GLOBAL.

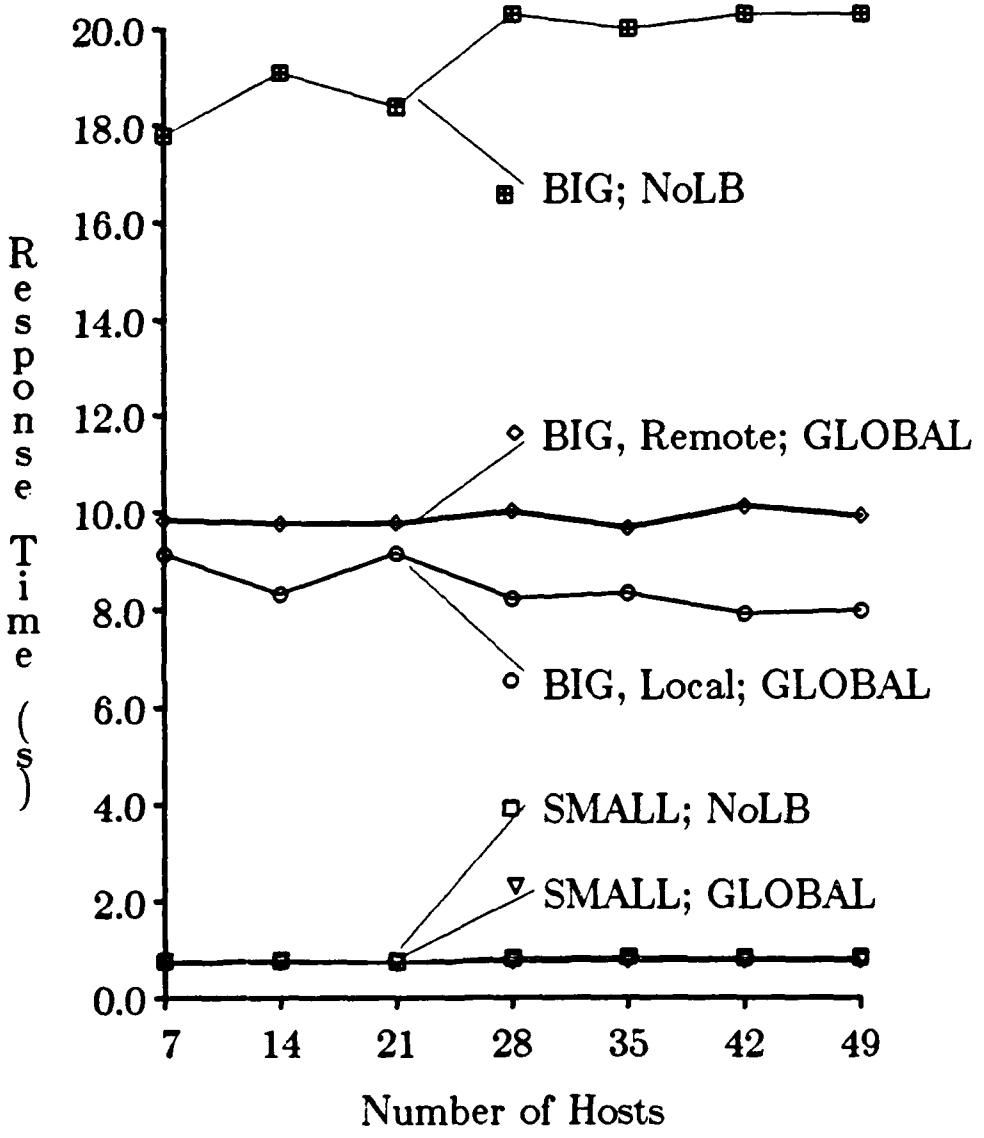

Figure 3.14. Average response times for several classes of jobs (28 hosts; GLOBAL, T_l=1.0, T_{CPU}=1.0s; utilizations same as in Figure 3.2).

As can be observed, the average response times of SMALL jobs under NoLB and GLOBAL are almost the same, whereas for BIG jobs the response time is roughly halved under GLOBAL. The average response time for Remote jobs is moderately higher than that for Local jobs, partly due to the placement and transfer overhead, and partly due to the fact that jobs tend to be transferred when system load is higher, thus causing higher response times. It is

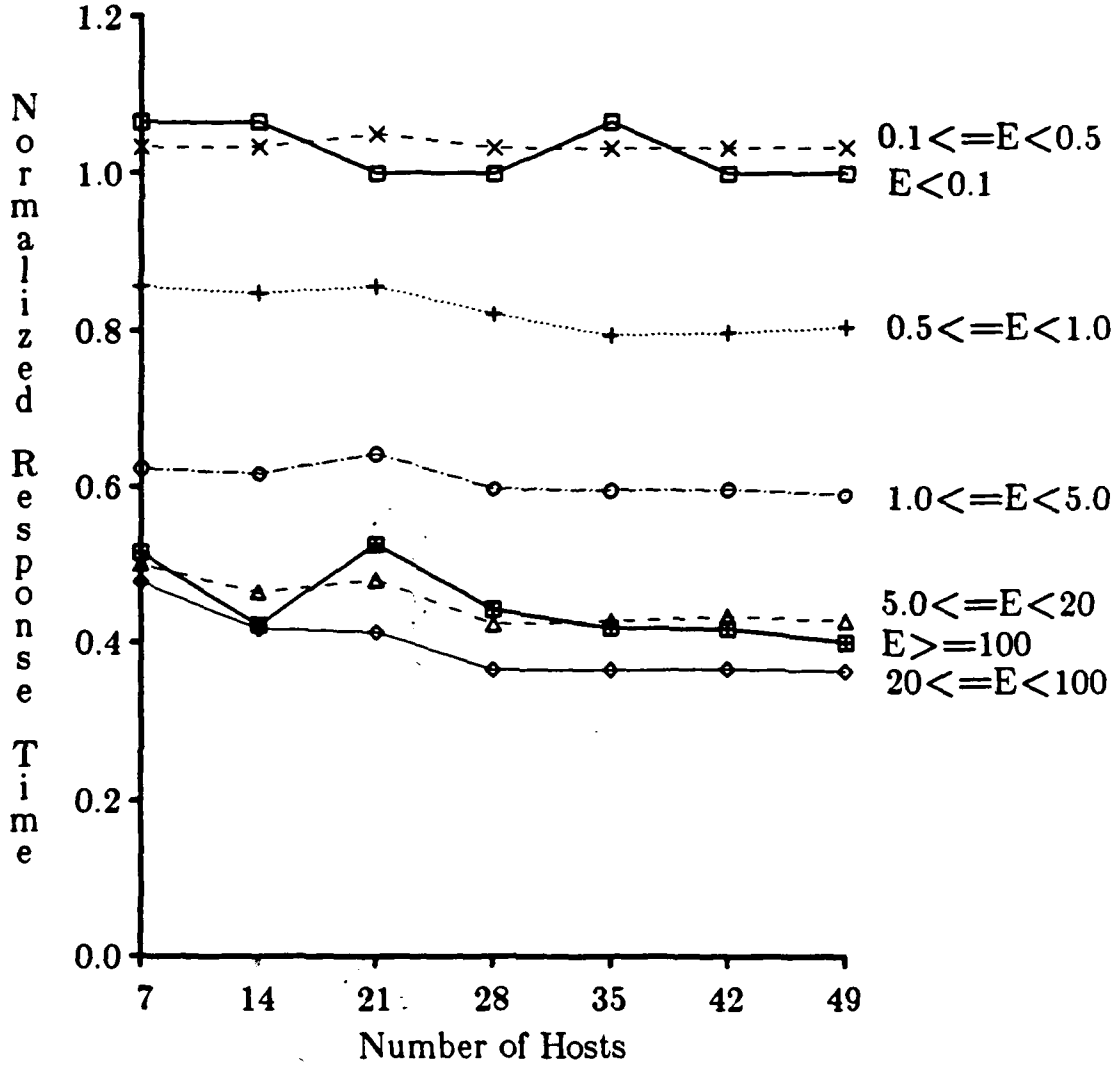

Figure 3.15. Average response times for several sizes of jobs (28 hosts; GLOBAL, T_l=1.0, T_{CPU}=1.0s; utilizations same as in Figure 3.2).

remarkable, however, that the Remote and Local jobs have comparable response times, implying that a user will not be penalized or rewarded much by having a job executed locally or remotely. The above comparisons are consistent throughout the range of system sizes. Simulations using other algorithms yielded similar results.

Figure 3.15 provides a more detailed picture of the performance improvement/degradation caused by load balancing to jobs with execution times, E, falling within each of the seven ranges. The average response times are normalized with respect to those with NoLB. The very small jobs ($E <$ 0.5s) suffer slightly. Such jobs are never transferred, and stay in the foreground

queue throughout their lives, so load balancing cannot reduce the competition among themselves. At the same time, load information exchanges and job placements have higher priority, causing them to wait. For all the rest of the jobs, however, the response times improve substantially, with bigger jobs showing greater improvements.

3.4.8. System Instability

The problem of the instability that may be introduced by load balancing is of major concern to the researchers in this field. It is feared that, because of the delay in load information exchange, several hosts may transfer jobs to a once lightly loaded host, and cause it to become overloaded. After the load information is updated, some other host(s) may in turn become the new victim(s). We call such phenomenon *host overloading*. Another form of instability is *job thrashing*, in which jobs are transferred too many times (or even for an indefinite number of times, as analytically shown in [Eager86b] for an algorithm similar to RANDOM) in an attempt to find an optimal host for job execution. Host overloading causes performance degradation because of unstable and uneven load distribution among the hosts, whereas, for job thrashing, degradation is mainly due to excessive job transfer overhead. Since we are mostly concerned with algorithms that transfer jobs only once, we will study only the host overloading problem here.

We consider a job transfer to be *wrong* if the destination host's CPU queue length is equal to or greater than that of the originating host. There is a distinction between transferring a job wrongly and collectively overloading a host; the former by itself will only increase the particular job's response time, whereas the latter will potentially cause system-wide performance degradation, due to the aggravated effects of the individual wrong transfers. This problem could be serious because usually the transferred jobs are big. To measure the level of host overloading occurring in a system directly is not easy; instead, we define the *host overloading factor* τ to be the percentage of wrong job transfers over all transfers, which may be regarded as a *pessimistic* approximation of the level of host overloading:

$$\tau = \frac{number\ of\ wrong\ transfers}{total\ number\ of\ transfers} \times 100\%$$

There are a number of factors that affect the rate at which wrong transfers are made. First, the staleness of load information has a deciding effect. The staler the information, the more the jobs that are transferred wrongly. Therefore, the non-periodic information policies that collect load information on demand are less susceptible to host overloading than the periodic policies. Another important factor is the rate at which jobs that are candidates for transfer arrive. This depends on the system load level and the job threshold.

The higher the load and the lower the job threshold, the larger the percentage of eligible jobs. To verify our intuitive argument, we calculated τ in simulation experiments for the GLOBAL algorithm using various values of the load exchange period, P, and the job threshold, T_{CPU}. Since it is difficult to consider three factors all changing at the same time, we fixed the system load level at 79%. Such a system-wide utilization is high, and host overloading may be expected to be quite serious. The results are shown in Figure 3.16, and agree with our reasoning above.

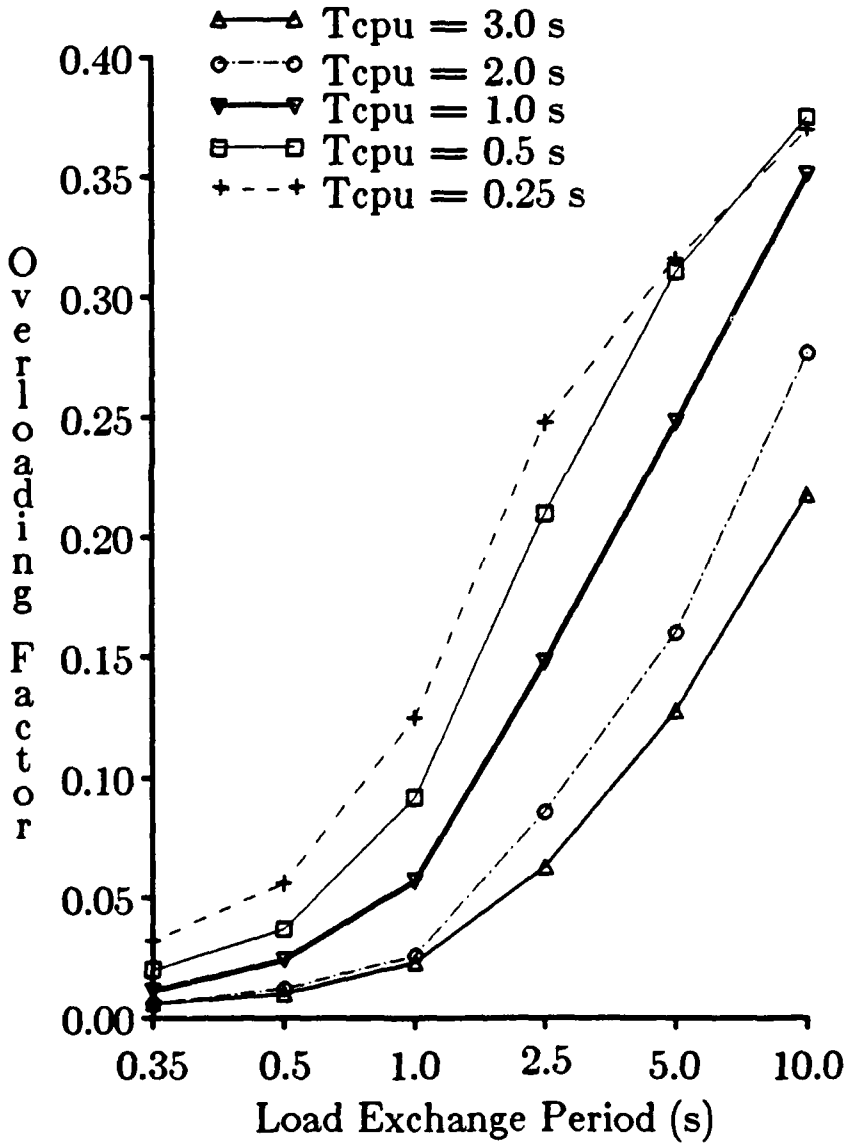

Figure 3.16. Percentage of wrong job placements for GLOBAL under various load exchange periods and job thresholds.
(number of hosts: 14, average utilization: 79%)

Besides the load update frequency and the job threshold, the system scale also affects host overloading, but to a lesser degree. It is important to know the number of hosts with the least load. For the algorithms studied in this chapter, placement decisions are based on the instantaneous CPU queue lengths of the hosts. Since there may be more than one host with the same shortest queue length, the transferred workload could be shared by them, thereby reducing overloading. A larger system size makes such situation more probable. On the other hand, in a larger system, there are also more sources of transferred jobs. To study quantitatively the number of hosts with the least load as a function of system size and load update period, we recorded the load vector at a high frequency during a simulation experiment for GLOBAL, and counted the number of hosts with the least number of jobs at their CPU's. The actual shortest queue length is unimportant because we are only concerned with the relative distribution here. Figure 3.17 shows the distributions for systems with 14 and 28 hosts, and the exchange period fixed at 5 seconds. For shorter exchange periods, the means of the number of hosts with the least load are slightly lower. We find that the probability of having only one or two hosts with the least load is non-negligible; hence host overloading can occur. Consider the following case of heavy load: for a system with 14 hosts and a load level of 80%, the total rate at which jobs are transferred by the GLOBAL algorithm using a job threshold of 1.0 second is in the range of 1-2 jobs/second. This means that, if we update the load information every 5 seconds, 5-10 jobs may be transferred to the single host that used to have the least load! This range is reduced to 1-2 jobs if the exchange period is 1.0 second, and even lower if the system load is not at such a high level. Hence, we see that whether host overloading occurs depends primarily on the system load level and the load exchange period.

From the above discussion, we can conclude that host overloading is possible with load balancing, especially when the system is heavily loaded, and stale load information is used for job placements. However, it seems to be easily avoidable by a number of simple measures, such as adjusting the algorithm's parameter values, and using algorithms that are less susceptible to it (e.g., THRHLD, LOWEST, and CENTRAL). The algorithms we studied show varying susceptibility to host overloading. GLOBAL and DISTED are the most susceptible ones, because they rely on locally maintained load information for job placements. In contrast, CENTRAL is much less susceptible because the LIC updates its load vector when job placements are made (see Section 3.3). THRHLD and LOWEST use host probing, hence rarely make wrong job transfers. An argument based on probability can be used to show that RANDOM is unlikely to cause severe host overloading. Even with GLOBAL, the average response times from simulation experiments corresponding to those in Figure 3.16 show that host overloading does not have as disastrous effects on system performance as we feared: very good performance can be achieved even

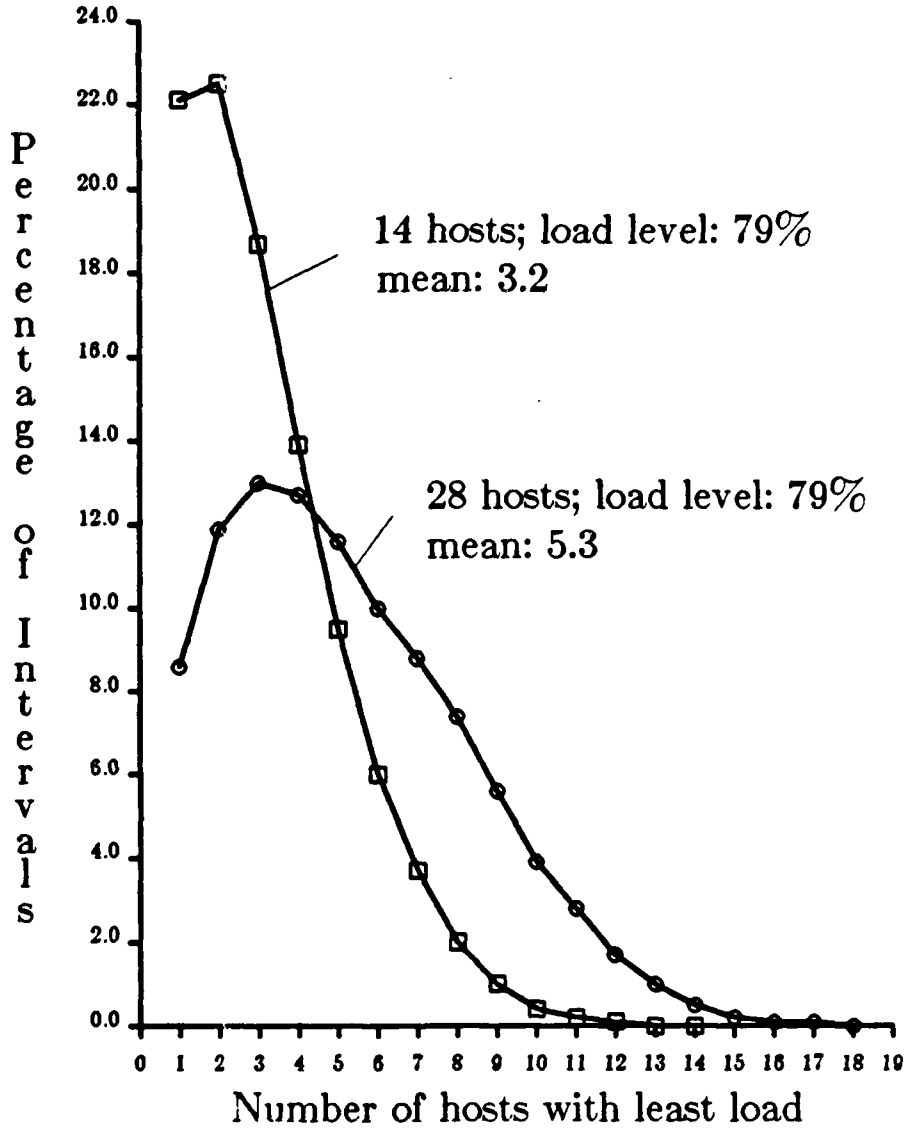

Figure 3.17. Distribution of the number of hosts with the least load ($P = 5.0$ s, $T_{CPU} = 1.0$ s).

when there exists light overloading ($\tau < 10\%$).

3.5. Summary

In this chapter, we studied dynamic load balancing using a simulation model driven by job traces collected from a production system. We simulated a representative CPU scheduling policy, and we considered explicitly the costs of load information exchange and job transfers. Because of the use of live system data, we believe that the results of our simulation are more reliable than those

from analytic models or simulations driven by probability distributions. On the other hand, we realize that our results may be biased towards a particular type of computing environment.

Seven load balancing algorithms were studied. We found that, under moderate system load, load balancing can reduce the average response time of all the jobs by 30-60%, and make them much more predictable. Algorithms using periodic and non-periodic information policies yield comparable performance; for the periodic load information policies, the centralized approach has much less overhead than the distributed approach, and, therefore, performs better and is more scalable. The only server-initiative algorithm we experimented with showed worse performance than the source-initiative algorithms, partially due to the restriction to initial job placement.

Some of the factors that affect load balancing performance were studied. We observed significant but limited economies of scale: performance improves as the number of hosts in the system increases, but, beyond a few tens of hosts, little further improvement results. Consequently, an algorithm with a scalability of up to 50-100 hosts seems to be sufficient. We found that performance improvement is greater under heavier system load, and that the adjustable parameters of the algorithms have varying effects on performance, suggesting that adaptive load balancing has potential. The impact of immobile jobs on load balancing was found to be less serious than the immobility factor might suggest: most of the performance gains are still retained even when up to 50% of the jobs eligible for remote execution are immobile. The performance of load balancing is found to be quite insensitive to the assumptions about the message and job transfer costs. The relative performances of the algorithms also stay the same under a wide range of overhead assumptions.

Load balancing has a profound impact on the system's behavior. In our simulations, the performances of *all* hosts, even those originally with lighter loads, improve under effective load balancing. This is somewhat counterintuitive, but very encouraging: by cooperating with each other, no one loses. We also observed a strong tendency of load balancing to equalize the loads, both long-term and short-term, of the individual hosts, While big jobs benefit more from load balancing, the response times of small jobs increase only slightly. For jobs of similar sizes, those transferred tend to have slightly longer response times. The problem of system instability due to several hosts sending jobs to the same host was studied. We found that such host overloading is possible, but can be effectively alleviated by using up-to-date load information in placement decision-making.

Chapter 4

Measurement Studies

4.1. Overview

The trace-driven simulation studies in Chapter 3 revealed some important properties of load balancing, and those of a number of algorithms. While the results are interesting and encouraging, we realize that the simple structure of the model and the simple representation of the jobs are bound to introduce errors in the results. Moreover, since only data from a particular environment were used in simulation, the results may be biased. In this and the next chapter, we will study load balancing in other environments and/or with other approaches in order to confirm and extend the previous results.

In this chapter, the design and implementation of a prototype load balancer in a loosely-coupled distributed system is discussed; furthermore, the results of a large number of measurement experiments performed on the system under the artificial workloads we constructed using frequently executed system commands are presented. The influence on the system's performance of the load balancing algorithms, as well as of the values of their adjustable parameters, and of the various types of workloads, is evaluated. The impacts of load balancing on the performance of individual hosts and on each type of job are also quantitatively investigated using measurement.

Our purposes in implementing a prototype load balancer and performing measurements on it were several-fold. First, we were interested in investigating the feasibility of load balancing, particularly in a UNIX environment. We wanted to experiment with various load indices proposed in the literature, in order to identify one or a family of load indices suitable for load balancing. We wanted to study, in a more realistic setting, the problems we had studied using simulation, in order to validate our results, and to use the simulator, with more confidence, in exploring parts of the design space unreachable by our measurements. We also wanted to assess quantitatively the amount of overhead introduced by the load information exchange and the job transfers between the hosts. Finally, we hoped to learn from our experimental implementation how to design a production load balancing system.

The important results of our measurement studies include the following:

- transparent, flexible load balancing at the job level can be achieved at a low cost, and without modifying either the system kernel or any of the existing application programs;
- load balancing is capable of substantially reducing the mean of the process response times (up to 30-40%), and their standard deviation (up to 40-50%), especially when the system is heavily loaded, and/or the instantaneous loads on the hosts are appreciably unbalanced;
- a number of "reasonable" load balancing algorithms using periodic load information exchange or acquiring such information on demand produce comparable performance improvements;
- load balancing can still be highly effective when only a small fraction of the workload (down to 20%, in terms of CPU time consumption) can be executed remotely;
- the relative (percentage) reduction in response time is uniform across all classes of jobs, mobile or immobile, large or small;
- load balancing at the job level has limited ability to reduce the temporal fluctuations in the load, mainly due to the generation of multiple processes by some single jobs;

The rest of the chapter is organized as follows. The design and implementation problems we dealt with are discussed in Section 4.2. In Section 4.3, we describe the design of the measurement experiments, including the artificial workloads we constructed. The results of the experiments are presented in Sections 4.4, 4.5, 4.6, and 4.7, with Section 4.4 comparing a number of load indices, Section 4.5 comparing the algorithms used for load balancing and assessing the importance of their adjustable parameters, Section 4.6 studying the effects of the workload on load balancing performance, and Section 4.7 discussing the impact of load balancing on individual hosts and job types. The major results are summarized in Section 4.8.

4.2. Design and Implementation

While the primary concerns of this dissertation are the performance issues that arise in load balancing, we are also very interested in studying the design and implementation of a load balancer. Such studies ensure that our performance work is of practical significance. The load balancer built as a result of our studies served as the basis of our measurements. The basic design and implementation of our prototype load balancer is presented in this section, followed by a description of the load balancing algorithms we have implemented and studied, and by some results of our overhead measurements.

4.2.1. System Basics

In Chapter 2, we surveyed the existing work on load balancing implementation and on remote execution facilities. While these load balancers have provided much knowledge about load balancing design and implementation, the requirements of our research were quite different from the ones of those systems. In designing our load balancer, we felt the following characteristics to be highly desirable:

1) *transparency*: no special syntax should be introduced, unless the user has some specific requirements; the placement of a job should be done automatically on the basis of the system's load conditions and the job's resource demands;

2) *no or little change to the system kernel*†: the cost of installing and maintaining the load balancer should be minimized;

3) *no modifications to commands and applications*: the code of no existing command should have to be modified to adapt it to load balancing;

4) *general applicability*: we are interested in considering *all* types of jobs, at least in principle, rather than only a specific category, e.g., text processing commands; also, the design should not assume any specific system architecture; the same design should be suitable for time-sharing systems and compute servers, as well as personal workstations, provided that certain basic requirements are met, namely, a communication system and the availability of a distributed file system.

Like the designers of the other load balancers, we were also concerned with the overhead of load balancing; the remote execution of a job should not incur high overhead in terms of extra processing or elapsed time delay. Since our implementation is experimental in nature, we are less concerned with issues such as remote process management and control, and user interface facilities.

There are two basic issues in the design of a load balancing system. The *policy* issue is concerned with the algorithm used to determine which jobs or processes should be executed remotely, and where. The *mechanism* issue is concerned with the physical facilities to be used for remote execution, i.e., with the way a job is transferred to a remote host and its results sent back. Before these two issues can be studied, however, we have to decide the level at which load balancing takes place. There are several choices. At the job, or command, level, the user interface can be changed so that some of the jobs submitted by

---

† In our implementation, we had to *add* a small amount of code to the system kernel to generate and maintain the load index used by the load balancing algorithms, and to provide enough precision for our measurements. No functional change, however, was made to the kernel.

the user may be redirected to some remote host for execution. Alternatively, load balancing can be done at the process level. In that case, the process management module of the system kernel must be modified to identify processes to be executed remotely. A third choice is to modify individual applications and incorporate remote execution facilities there [Johnston86]. However, considering our requirements discussed above, the second and third approaches are to be ruled out.

After the level of load balancing is determined, we still have to decide whether the jobs or processes are to be transferred during their execution (process migration), or only at start-up time (initial placement). Process migration has been suggested by a number of researchers as potentially more capable of improving system performance [Leland86] [Cabrera86]. On the other hand, it is also likely to incur higher overhead, and is very difficult to implement in such systems as UNIX. In addition, since we decided to do load balancing at the job level, and multiple processes may be created by a single job, we would have to consider the interactions between the processes explicitly. These considerations led us to restrict ourselves to initial job placement in our experimental load balancer.

Our implementation is based on a modified C shell† implemented at Berkeley by Venkat Rangan and Harry Rubin for the Berkeley UNIX 4.3 BSD system running on VAX machines [Joy83] [McKusick85]. This modified C shell intercepts user commands and executes certain types of commands remotely when the local host is heavily loaded, using the *rexec* daemon available in the system. The structure of our system is depicted in Figure 4.1‡. At startup time, the C-shell reads in a configuration file that specifies a list of names of jobs that are eligible for remote execution*. When an eligible job is submitted by the user to the C-shell, the C-shell contacts a *Load Information Manager* (LIM), a software module that constantly monitors the loads of the hosts in the system and performs job placements. If the initial host is heavily loaded, while some other hosts are not, one of the remote hosts is selected as the destination for the job. In any case, the placement decision is returned to the C-shell. For remote execution, the C-shell contacts the *Load Balance Manager* (LBM)** on the destination host, which starts up an R-shell (i.e., creates a process running the R-shell

---

† C shell is the name of the command interpreter in the Berkeley UNIX operating system [Joy80].

‡ To distinguish our modified C shell from the standard one, we call it *C-shell*. The *R-shell*, to be described below, shares the same software with C-shell, but is only to receive remote jobs and execute them.

* This list is part of the context of each user, just like command aliases, and may be dynamically modified by the user to suit his or her needs.

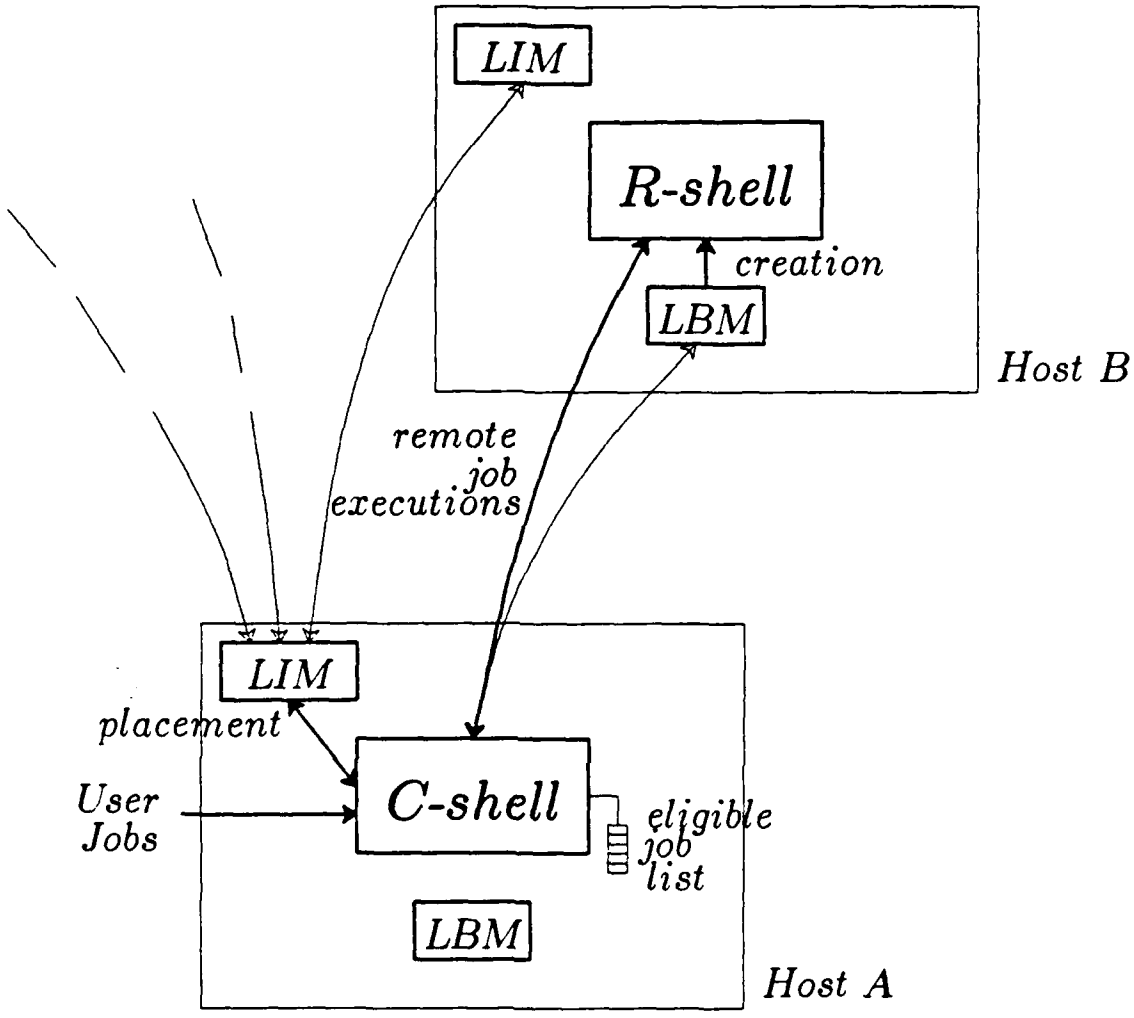

Figure 4.1. Structure of load balancing implementation.

program) and establishes a stream connection between it and the home C-shell. The command line is transmitted over this connection to the R-shell after the user's identity has been authenticated, and an appropriate user environment has been set up there. Access control to files and other resources in the system is automatically enforced as the R-shell assumes the same user identity as that of the home C-shell. Since starting an R-shell is an expensive operation, as we will see below, we keep such a shell alive after the execution of the first job so that if a later command from the same user login session is placed on the same host, we do not have to go through the same process described above. The R-shells on

** Note that there is one LIM and one LBM on each host.

remote hosts act as agents for the home C-shell, and are terminated when the home C-shell exits. This scheme has the potential problem of a proliferation of R-shells. However, the code segments of all C-shells and R-shells on each host are shared, so that, when an R-shell is not active, almost no resources are consumed by it.

Thus, our load balancing system design is at the job level, and stresses a clear separation of policy from mechanism. The collection and management of load information, and the job placement decision-making are performed by the LIMs, one on each host, and cooperating among themselves in ways dictated by the load balancing algorithm. Only the initial filtering of jobs by their names is performed in the C-shell to avoid querying the LIM too frequently, and to allow personalized selection of jobs. The load balancing mechanism, on the other hand, is provided by the LBM on each host, with the cooperation of the C-shell and R-shell. The separation between policy and mechanism makes it easy to experiment with different algorithms, as only the LIM needs to be changed. In fact, the LIM software can be constructed so that the load balancing algorithm may be changed dynamically as the system's size and load change.

We assume that the distributed system includes a distributed file system supported by one or more file servers. Consequently, the program and data files of a job do not have to be fetched from the originating host, but from a file server, no matter where the job is executed. Thus, we assume that the cost of accessing the files is the same for all hosts. While this is true for diskless workstations supported by file servers, the location of the program and data is an important factor to consider in systems where files are scattered on a number of hosts (which are not dedicated file servers). We decided not to consider this problem in order to concentrate on the issues we are most concerned with now.

Two slightly different implementations were built, one for the Sun/UNIX system, one for the Berkeley UNIX 4.3 BSD system. While the first is fully operational, and was used for the measurements, the second is not because a distributed file system is not supported by the operating system yet.

4.2.2. Algorithms

A large number of algorithms have been proposed in the literature (see [Wang85] for a taxonomy). The problem domain we are concerned with in this research (i.e., initial job placement in a loosely-coupled network environment for general-purpose computing, with distributed job submissions), and our desire to *implement* the algorithms make many of the proposed algorithms unsuitable. While all of the seven algorithms studied in Chapter 3 could be implemented, we selected five of the more representative ones, GLOBAL, DISTED, CENTRAL, RANDOM, and LOWEST, for implementation. In addition, the NoLB case is used for comparison. Instead of repeating the descriptions of the algorithms in

Section 3.3, we will only mention the adaptations that were necessary for their implementation.

The LIMs on the hosts exchange load information among themselves according to the algorithm adopted. For example, in GLOBAL and CENTRAL, one of them is designated as the master (just as the LIC in the simulation), and collects load information from every host. In CENTRAL, the master also makes all the job placements. When the non-periodic algorithm LOWEST is used, however, the LIM on the host where a job just arrived will probe a few other hosts.

For the common job transfer policy, we still use the local load threshold T_l, but not the job execution time threshold T_{CPU}, as, in reality, it is not known upon the job's arrival. Instead, the list of job types eligible for load balancing in the configuration file is consulted. This may be regarded as a very rough approximation of the job threshold, and is certainly much more realistic.

Some refinements to the basic algorithms have been implemented. For example, in the periodic algorithms, the local load is sent out only if the new value of the load index is significantly different from the previous one reported. This is found to reduce the message traffic by 50-70%, without affecting the quality of the job placements much. This observation agrees with a similar one made in simulation (see Section 3.4.1). The local load is not reported if it stays above an upper threshold T_u. However, the local load is reported once in a while even if the it has not changed much, so that other hosts will not assume that this host is unavailable.

For the meanings of the symbols representing the parameters of the algorithms and other quantities, please refer back to Table 3.2.

4.2.3. Overhead Assessment

We measured the additional CPU processing and job delays due to load balancing, that is, to the exchanges of load information, the job placements, and the remote executions. Table 4.1 shows some of the results for Sun-2 workstations with 2 MB of memory and a 3Com Ethernet board. Note that all times in the table are real time delays averaged over a few hundred to a few thousand repetitions. The numbers differ somewhat from those used in the simulation because the latter were obtained in a VAX time-sharing environment. The measurements were taken on empty hosts. When the system is loaded, the delays become longer and their variance increases. For locally executed jobs, the average overhead is very low, typically 5-10 milliseconds, and is mainly due to searching the job list in the C-shell, and, if the job name is on the list, to querying the LIM. The delay due to a LIM query plus the overhead of remote execution is highly variable, depending on the loads of source and destination hosts. On the average, it is a few hundred milliseconds. This assumes that an R-shell has already been set up on the destination host. Otherwise, several seconds of

Table 4.1. Load balancing overhead measurements.

extract load info. from kernel and send out a message (500 bytes)	14.5 ms
receive a load message and store it into load vector (500 bytes)	5.7 ms
placement request by C-shell to LIM (round-trip)	
to local LIM	23.8 ms
to remote LIM (for CENTRAL)	52.9 ms
remote job execution overhead (incl. placement by local LIM, assuming R-shell already set up)	325 ms
start an R-shell (setup)	5 s

additional delay may be incurred. Overall, the overhead of load balancing seems to be quite low. With an exchange period of 3 seconds, load information updates cost from one to a few percent of Sun-2 CPU time. The delay due to remote execution is hardly perceivable by an interactive user, and is very small compared to the average job response time, which is of the order of a few tens of seconds.

4.3. Experiment Design

4.3.1. Performance Index

As in our simulation studies, the mean response time of all the jobs executed during a measurement session, supplemented by the standard deviation of the job response times, seems to be an appropriate performance index. However, the response times of jobs executed remotely in the background turned out to be difficult to obtain in our implementation. Instead, we made use of the system accounting facility to obtain the response times of all the *processes* executed during a measurement session, and used the mean process response time as our performance index. For the execution of most of the jobs, only one process is created, so the two indices are the same, except for the command line processing in the C-shell, which is not accounted for in the process response time. For a few commands (namely, *cc, lint*, and *ditroff in our artificial workload*), however, several processes are created, and their response times are all considered in computing the mean. The overhead of load balancing is accounted for by measuring it during each experiment run and adding it to the process response times.

4.3.2. Experimental Factors

We identify four major factors that affect the performance of a load balancing system. First, load indices that capture the current load conditions and are capable of predicting host load in the near future are of crucial importance. A poor load index may cause job transfers that do not contribute to balancing the load of the system, and might even make things worse. Secondly, the algorithm used for load balancing determines the cost of distributing load information, and the quality of job transfers. Thirdly, the performance improvements due to load balancing are dependent also on the workload the system is subjected to. The workload will be characterized along two dimensions, which will be considered as independent factors: that of its *intensity*, i.e., its magnitude, and that of its *mobility*, i.e., the fraction of the workload (as defined in Section 4.6.2) that can to be executed remotely. Lastly, the underlying implementation of the load balancer certainly impacts load balancing performance, but since the implementation is fixed in our case, our measurement experiments only explore the remaining dimensions. More specifically, we vary one factor at a time and study its influence. A number of levels or values are assigned to each of the factors, as listed in Table 4.2.

Table 4.2. Experimental factors and their levels.

Load index:	CPU utilization; instantaneous CPU queue length; time-averaged CPU queue length; linear combination of averaged CPU, paging/swapping, and I/O queue length; load average (see Section 4.3.3)
Algorithm:	NoLB, DISTED, GLOBAL, CENTRAL, LOWEST, RANDOM each algorithm has a number of adjustable parameters (see Section 4.2.2)
Workload intensity:	canonical workload: <2H, 2M, 2L>; balanced workloads: <5H>; <6M>; <6L> (see Section 4.3.4)
Workload mobility:	several values of the *immobility factor* (see Section 4.6.2)

In the next two sections, the load indices and the types of workload will be described.

4.3.3. Load Index

The survey of load indices in Section 2.6 shows that there are a large number of possible indices, and, conceivably, load balancing performance is affected by the load index. Since it is impossible to study all indices, we compare two families of indices using measurement techniques. One family is based on the utilization of the CPU. The other is that proposed by Ferrari, as a linear combination of resource queue lengths. Since the amounts of resources required by jobs are difficult to predict, we will only evaluate those indices whose coefficients of the resource queue lengths are *job independent*, and only reflect the relative importance of the resources (with respect to a "basket" of jobs). For example, we can use unity as the coefficients to reduce the linear combination to the sum of the resource queue lengths, that is, in queueing modeling terms, to "the number of jobs in the system."

The resource queues we considered are as follows:

1) CPU ready queue: the number of processes running, or loaded and ready to run;
2) file I/O queue: the number of processes waiting for file I/O on disks to complete;
3) paging/swapping I/O queue: the number of processes waiting for a page, or being swapped in/out;
4) memory queue: the number of processes waiting for various types of memory resources (e.g., buffer space, page table).

Our extensive measurements of production time-sharing workloads show that the system load is changing quite rapidly [Zhou87a]. On top of a low-frequency main component, there are a number of high-frequency load components that may be regarded as "noise" rather than useful information. Using the instantaneous resource queue lengths may give excessive importance to such noise, and lead to bad job transfer decisions. We used a smoothing algorithm to compute the time-averaged queue length, and compared load balancing performance using smoothed queue lengths to that of the same scheme using instantaneous queue lengths. Inside the kernel, we kept variables for the queue length of each of the four resource types above. The length of each queue was sampled every 10 milliseconds in the clock interrupt routine, and used to compute the one-second average queue length, q_i, over the 100 samples. Exponential smoothing was then used, to compute the average queue length over the last T seconds:

$$Q_i = Q_{i-1}(1-e^{-T}) + q_i e^{-T}, \qquad i \geq 1$$

$$Q_0 = 0$$

By changing the value of T, the range of averaging can be adjusted. In our measurements, the sum of the queue lengths of some or all of the four types of

resources, smoothed over 1, 4, 20 or 60 seconds, are used and their performances compared.

4.3.4. Workload

The construction of workloads accounted for most of our efforts in the design of the experiments. On the one hand, since a high degree of repeatability of the experiments was felt to be absolutely necessary, we used artificial workloads. On the other hand, we wanted these workloads to represent real workloads reasonably well, so that we could have confidence in the realism of the results. We traced a production VAX-11/780 machine running under the Berkeley *UNIX 4.3BSD system* [Joy83] [McKusick85] for an extended period of several months, and analyzed the types and frequencies of the commands executed by the system. On the basis of such an analysis, we selected a number of frequently executed commands, as listed in Table 4.3, and used them to construct scripts, i.e, streams of commands. To obtain various levels, or intensities, of load, such as those characterizing multi-user systems, we ran a variable number of the jobs in the background. Also, we simulated user think times by the "sleep" command. The scripts are classified into three levels: light (L), moderate (M), and heavy (H), with a number of distinct scripts constructed for each level so that hosts subjected to the same level of workload can use different scripts. The ranges of CPU utilizations and mean load index values of the three levels of scripts are shown in Table 4.4. Each script runs for about 30 minutes on a Sun-2 workstation. Job and system performance statistics, such as resource demands, response times, resource utilizations, and resource queue lengths, were measured throughout each run.

A system workload is a combination of the host workloads. So, for a system of six hosts, we define the *canonical workload*, intended to be a "typical" loading situation, as <2H, 2M, 2L>, that is, two workstations driven by heavy loads, two by moderate loads, and two by light loads. We also used more balanced system workloads, e.g., <5H>, <6M>, and <6L>.

As in any measurement experiment, we must consider the variability of the experimental environment, and, therefore, that of the measurement results. In dynamic load balancing, the placement of each job may vary from one run of the experiment to the next, because of the unavoidable variations in the timings of the events. (This problem was further complicated in our experiments by the fact that we had to share the file server and the network with other parts of the research community. We tried to minimize this impact by running the experiments during the night.) Thus, we repeated the same experiment a number of times (typically 6), and computed the mean and the 90% confidence interval (CI) of the performance indices over these replications.

Table 4.3. Commands used in scripts and their eligibilities for remote execution.

cmd.	**elig.**	**function**	**cmd.**	**elig.**	**function**
cat	N	*view a file*	*ls*	N	*directory listing*
cc	Y	*C compiler*	*man*	Y	*manual page viewing*
cp	N	*file copying*	*mv*	N	*move a file*
date	N	*current time*	*nroff*	Y	*text formater*
df	N	*file system usage*	*ps*	N	*process checking*
ditroff	Y	*text formater*	*pwd*	N	*current directory*
du	N	*disk usage*	*rm*	N	*delete a file*
egrep	Y	*text pattern search*	*sort*	N	*file sorting*
eqn	Y	*equation formater*	*spell*	Y	*spelling checker*
fgrep	Y	*text pattern search*	*tbl*	Y	*table formater*
finger	N	*user information*	*troff*	Y	*text formater*
grep	Y	*text pattern search*	*uptime*	N	*system uptime*
grn	Y	*graph printing*	*users*	N	*list of current users*
lint	Y	*C program checker*	*wc*	N	*word count in a file*
lpq	N	*printer queue check*	*who*	N	*user information*

Table 4.4. Characterization of the workload levels.

type	CPU utilization	average CPU queue
light	30-45%	0.3-0.7
moderate	60-70%	1.0-1.8
heavy	70-85%	1.8-3.0

4.4. Comparison of Load Indices

We shall study the indices and the averaging interval T by fixing the workload at its canonical level, the algorithm to be GLOBAL, and the load exchange interval P at 10 seconds. We shall then use the more balanced workload <6M> to examine the interactions between load indices and workload.

4.4.1. Canonical Workload

Table 4.5 shows the performance of the experimental system under various load indices. The numbers following the response time values indicate their 90% confidence intervals.

We see in Table 4.5 that all the indices provide performance improvement, that is, they all contain some amount of *current* load information. The amount of improvement, however, varies quite widely: from 20% to 40%. This means that the performance of load balancing is indeed heavily dependent on the load index used, and hence studying load indices is important. Comparing the two families of indices, those based on resource queue lengths are able to perform substantially better than those based on CPU utilization. This is probably because, when a host is heavily loaded, its CPU utilization is likely to be close to 100%; thus, in that region, the exact load level cannot be reflected by the value of the utilization. In contrast, queue lengths can directly reflect the amount of contention for a resource under heavy load. As an example, both a resource with an average queue length of 3 and one with a queue length of 6 probably have utilizations close to 100%, while they are obviously very differently loaded.

Comparing the queue-length-based indices with each other, we notice that the exponentially smoothed indices can perform better, but, if the averaging period T is too long (e.g., $T \geq 20$ s), performance may even become worse. Earlier in this chapter, we have pointed out that, by averaging the queue lengths, the adverse effect of the high-frequency "noise" in the load can be reduced. This is reflected by improved performance. However, since the system load is changing all the time, averaging over too long a period will emphasize too much the past loads, which have little correlation with the future ones. The optimum averaging interval is clearly dependent upon the dynamics of the workload: the faster the load changes, the shorter the interval should be. In a measurement study of production workloads on a VAX-11/780 running Berkeley UNIX 4.2BSD [Zhou87a], we found that the average *net change* in CPU queue length in 30 seconds was 2.31, when the average CPU queue length itself was 4.12. This suggests that, in that environment, T should be substantially shorter than 30 seconds.

---

† All confidence intervals in the tables and figures of this chapter have been computed with a 90% confidence level.

Table 4.5. Measured performance with various indices.
(Canonical workload, $P = 10$ s)

replication count: 6
total number of jobs per run: 501
total number of eligible jobs per run: 254 (50.7%)
total number of processes per run: 766 (1.53 processes/job)
average process execution time: 7.45 s
approximate average CPU utilization for NoLB case: 60%

Load Index	Resp. Time	Improv.	Std. Dev.	Improv.
NoLB (no load bal.)	53.3 ±0.83	—	90.1	—
inst. CPU ql	35.0 ±0.68	34.4%	46.7	46.7%
1 s avg CPU ql	33.8 ±0.65	36.6%	45.8	49.2%
4 s avg CPU ql	33.1 ±0.39	37.9%	42.3	48.7%
4 s CPU+I/O+Mem ql	32.2 ±0.45	39.6%	44.3	50.9%
20 s avg CPU ql	37.0 ±1.20	30.6%	51.8	42.6%
20s CPU+I/O+Mem ql	35.6 ±0.12	33.3%	49.1	45.6%
60 s avg CPU ql	39.7 ±1.69	25.5%	54.1	40.0%
60s CPU+I/O+Mem ql	40.0 ±0.56	25.0%	56.2	37.6%
60 s UNIX load average	37.2 ±0.85	30.2%	54.9	39.1%
10 s CPU utilization	38.5 ±2.10	27.8%	55.4	38.5%
60 s CPU utilization	42.9 ±1.36	19.5%	67.6	25.0%

The performance difference between the cases in which indices based on CPU queue length alone are used, and those in which indices consider I/O and memory contention also, is not significant, suggesting that the CPU is the predominant resource in our hosts. We found that the I/O and memory queue lengths were generally much shorter than that of CPU; that is, the former resources are much less contended for. It should be pointed out that our systems support general computing in a research environment; with other types of

workload, e.g., database-oriented ones, the contention profile of the various resource types may be substantially different. However, to achieve near-optimal performance, we do not have to consider all the resources in the system, but rather only those with significant contention. We also studied more general forms of linear combinations of queue lengths by using coefficients other than unity, but no significant changes in performance were observed. This, again, is certainly due to the dominating influence of the CPU queue.

The load average shown in Table 4.5 has been used in a number of load balancers constructed in the past in the UNIX environment (e.g., [Hwang82] and [Bershad85]). This research shows that significant further improvement can be obtained by using indices that more accurately reflect the current queueing at the resources.

4.4.2. Moderate, Balanced Workload

The performances obtained using various indices under the more balanced workload <6M> are shown in Table 4.6.

Table 4.6. Measured performance with various indices.
(<6M> workload, $P = 10$ s)

Load Index	Resp. Time	Improv.	Std. Dev.	Improv.
NoLB	49.5 ±0.27	—	72.4	—
inst. CPU ql	42.3 ±0.79	14.5%	61.4	15.2%
4 s avg CPU ql	39.9 ±0.63	19.4%	54.0	25.4%
4 s CPU+I/O+Mem ql	36.5 ±0.91	26.3%	51.0	29.6%
20s CPU+I/O+Mem ql	45.2 ±0.89	8.7%	63.8	11.9%
60s CPU+I/O+Mem ql	47.1 ±1.34	4.9%	67.7	6.5%
60 s load average	47.9 ±1.12	3.2%	73.1	-1.0%
10 s CPU utilization	44.0 ±1.97	11.1%	60.9	15.9%
60 s CPU utilization	48.6 ±1.34	1.8%	68.3	5.7%

Since the workload is now more balanced and moderate, the amount of improvement in response time is not as much as that under the canonical workload; however, the relative rankings of the indices are quite similar. This suggests

that the above analyses of the qualities of the indices and the appropriate values for T remain valid under a more balanced, moderate workload. It is worth noting that, in this case, due to the smaller improvement, using a poor load index (e.g., the load average or the 60 s CPU utilization) may yield little or no performance improvement.

4.5. Comparison of Algorithms

We first compare the performances of the algorithms, then study the effects of their adjustable parameters. For this and all the following sections, the load index is fixed to be the sum of the CPU, file I/O, and the paging/swapping queues.

4.5.1. Basic Comparisons

To compare the performances of the five algorithms described in Section 4.2.2, we applied each of them to a system of six Sun-2 workstations running the canonical workload described in Section 4.3.4. For each of the algorithms, we varied the adjustable parameters (considered as secondary factors), such as the local load threshold T_l, the load exchange period P for the periodic policies, and the probe limit L_p for the non-periodic policy (LOWEST), in order to achieve the *best* performance under that algorithm. For each algorithm, Table 4.7 shows the mean response time and its 90% confidence interval, the percentage improvement in response time relative to the NoLB case, the standard deviation of the response times and its percentage improvement, and the values of the adjustable parameters used in the run.

Table 4.7. Performance of the algorithms
(canonical workload; all times are in seconds).

Algorithm	Resp. Time	Improv.	Std. Dev.	Improv.	Parameters
NoLB	53.3 ±0.83	—	90.1	—	—
DISTED	36.4 ±0.09	31.7%	50.6	43.8%	P=15, T_l=0.8
GLOBAL	32.6 ±0.67	38.9%	43.6	51.7%	P=5, T_l=0.8
CENTRAL	33.7 ±0.54	36.8%	48.5	46.8%	P=10, T_l=0.8
LOWEST	31.8 ±0.37	40.3%	42.8	52.5%	P_r=4, T_l=0.8
RANDOM	39.9 ±1.21	25.2%	62.0	31.2%	T_l=0.8

The first observation one can make about the results in Table 4.7 is that load balancing can indeed improve system performance substantially. The canonical workload was constructed to reflect a loading situation commonly observed in production environments: some workstations are loaded, while others are not. By transferring jobs from heavily loaded hosts to lightly loaded ones, the mean job response time can be improved. Comparing the improvements in mean response time and those in the standard deviation of the response times, we notice that the latter is reduced more substantially. This means that the job response times are more predictable with load balancing than without.

The performances of the algorithms, except those of RANDOM and DISTED, are quite close to each other. For the periodic algorithms, the information is ready when a job is to be placed, and the "best" host in the system is selected. However, the periodic updates incur higher computation and communication overhead than the polling method used by LOWEST, and the load information used for placements tends to be less current than that in LOWEST. Comparing DISTED and GLOBAL, we see the adverse effect of the excess use of broadcast messages, as the two algorithms are the same except that, in GLOBAL, a master is used to collect and distribute load information. As a result, only the master has to handle N messages per period P, where N is the number of hosts, while all the other hosts need only to send one message and receive one during each period.

It should be pointed out that our measurement results and their analyses are in excellent agreement with those from simulation (see Section 3.4.1), despite the differences in computing environment and in workload, and the experimental errors.

A complete evaluation of the qualities of the algorithms cannot be done using a system of only six hosts. However, since our measurement results agree well with those of our simulations, they may be viewed as validation of the latter. Consequently, our simulator may be used, with more confidence, to explore parts of the design space unreachable by measurement.

4.5.2. Adjustable Parameters

The performance of load balancing is dependent on the parameter values used in the algorithms. While it is impractical to explore all the possible variations, or even to present here all the experiments we performed, we show the effects of the three most important parameters, namely, the load exchange period P, the local load threshold T_l, and the probe limit L_p, on three of the algorithms, GLOBAL, RANDOM, and LOWEST, respectively. For all cases, the canonical workload is applied to the six-host system, and the brackets around the data points show the 90% confidence intervals.

The mean process response times of GLOBAL using various values of P are shown in Figure 4.2.

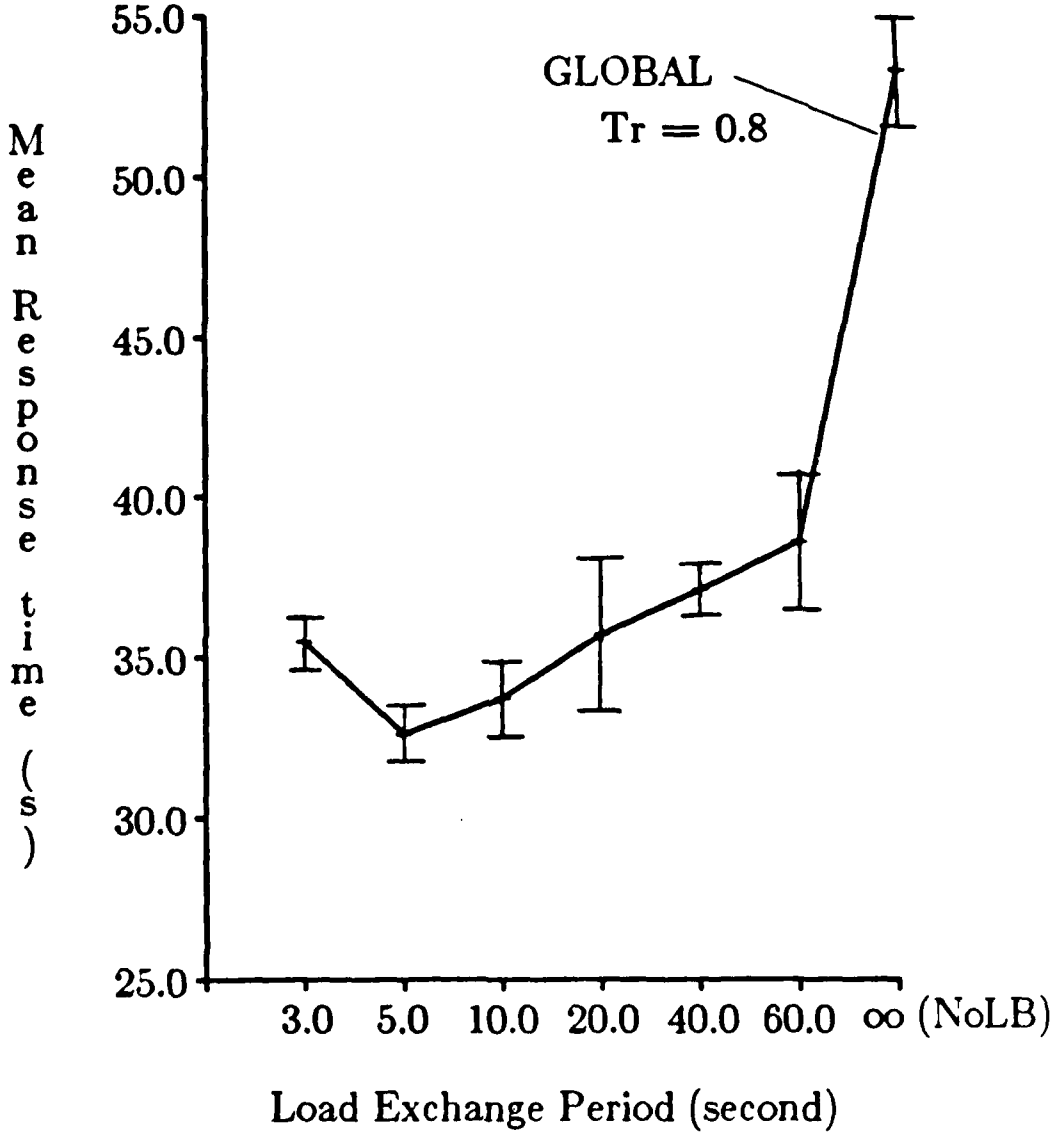

Figure 4.2. Mean process response time under various load exchange periods P (Canonical workload, GLOBAL, T_l=0.8).

When the exchange rate is too high, the overhead outweighs the benefit of up-to-date information. On the other hand, if the rate is too low, the information may get too stale, and performance suffers. The optimal exchange rate is also dependent on the workload. Specifically, the rate should be high if the job arrival rate is high and the average resource demands of the jobs are low. This is the case in our simulation studies for multi-user time-sharing systems. (The optimal exchange period in Figure 3.4 is 3 seconds, instead of 5 seconds in Figure

4.2.) Comparing Figure 4.2 to Figure 3.4, we found a similar pattern in the curves. However, the simulation curves have steeper slopes, implying that performance is more sensitive to P. This may be due to the higher rate at which the load fluctuates in simulation. It is remarkable that substantial performance gains are still achieved with an exchange period as long as 60 seconds. At that point, host overloading inevitably occurs. The message here seems to be that a load balancing system can tolerate a certain level of host overloading without suffering substantial performance degradation.

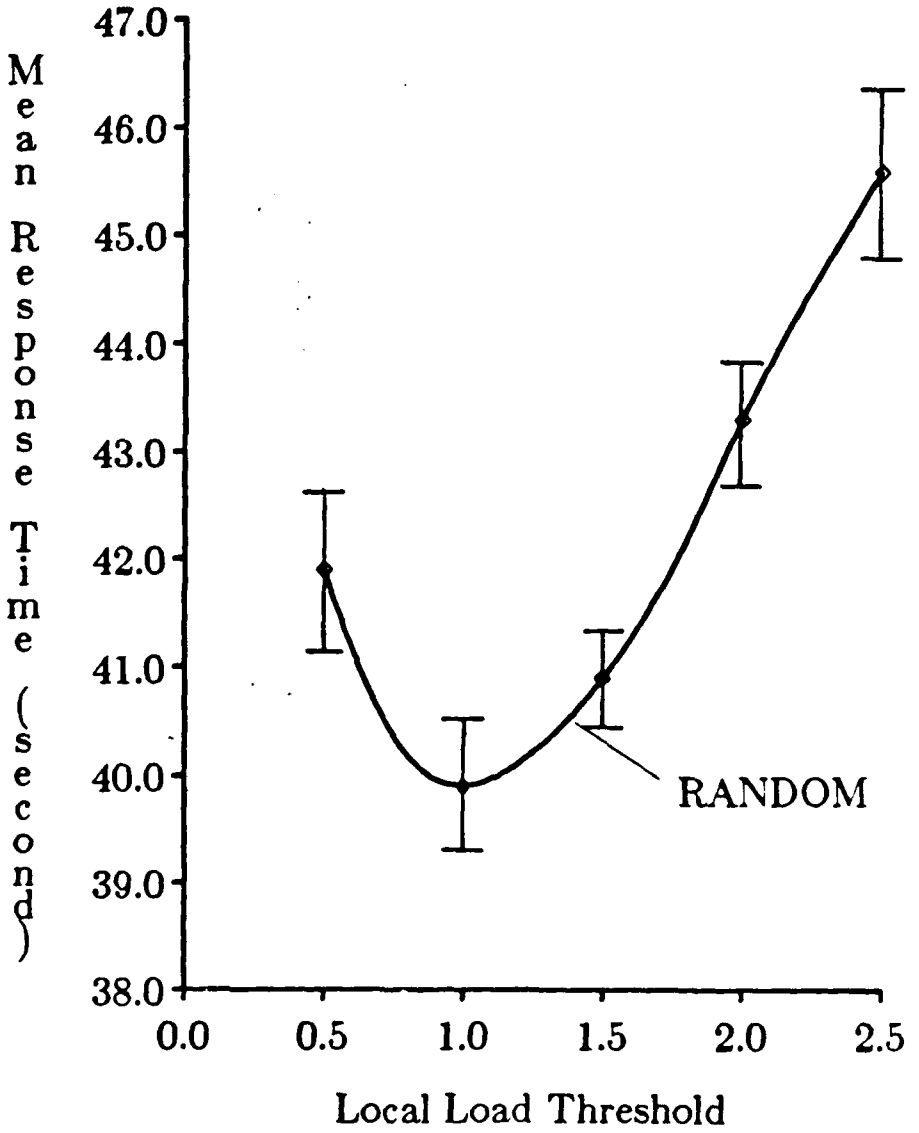

Figure 4.3. Mean process response time under various local load thresholds T_l (Canonical workload, RANDOM).

Similarly, there are conflicting requirements for the local load threshold T_l. On the one hand, a sufficient number of jobs have to be transferred between the hosts in order to balance their loads. On the other hand, however, an excessive amount of job transfers will increase system overhead, and may even cause severe host overloading. This tradeoff is illustrated by Figure 4.3, which shows the relationship between the mean response time and the local load threshold for the RANDOM algorithm, which uses T_l as its sole parameter. Again, the optimal threshold is dependent on the load level of the system. If all the hosts are subjected to heavy workloads, T_l should be set relatively high to avoid unproductive job transfers. As we observed in Figure 3.4, performance is not very sensitive to T_l within a range. When T_l changes from 0.5 s to 2.0 s, the mean response time stayed between 40 and 43 s.

We also studied the performance of LOWEST with various values of the host probe limit L_p. The results are displayed in Figure 4.4, which shows a minimum like those in Figures 4.2 and 4.3. Compared to Figure 3.5, performance also improves as multiple hosts are probed, but the upward turning point is lower in measurements, at 4 hosts. At this point, however, almost all the hosts in the system are probed. With the two systems drastically different in size (6 vs. 28), fine comparisons are difficult.

From Figures 4.2, 4.3, and 4.4, it is clear that the parameter values of the algorithms should be dynamically and automatically adjusted as the system load conditions change over time, in order to keep obtaining most of the performance gains of load balancing. This supports the proposal of adaptive load balancing mentioned in Chapter 3.

4.6. Performance under Different Workloads

The previous section compared the performances of the five chosen algorithms using the canonical workload. In this section, we study load balancing performance under different workloads. We first study workloads of different intensities, then study those with different levels of mobility. The GLOBAL algorithm, which demonstrated good performance, was chosen for this part of the study.

4.6.1. Different Intensities

Tables 4.8, 4.9, and 4.10 show the values of the performance indices and their improvements relative to the NoLB case when all hosts in the system are subjected to heavy, moderate, and light load, respectively. Although the load level is the same for all the hosts, separately constructed scripts are used so that no synchronization effect will occur.

For the canonical workload studied in the previous section, significant differences in host loads over the entire run (*long-term* imbalances) exists, hence

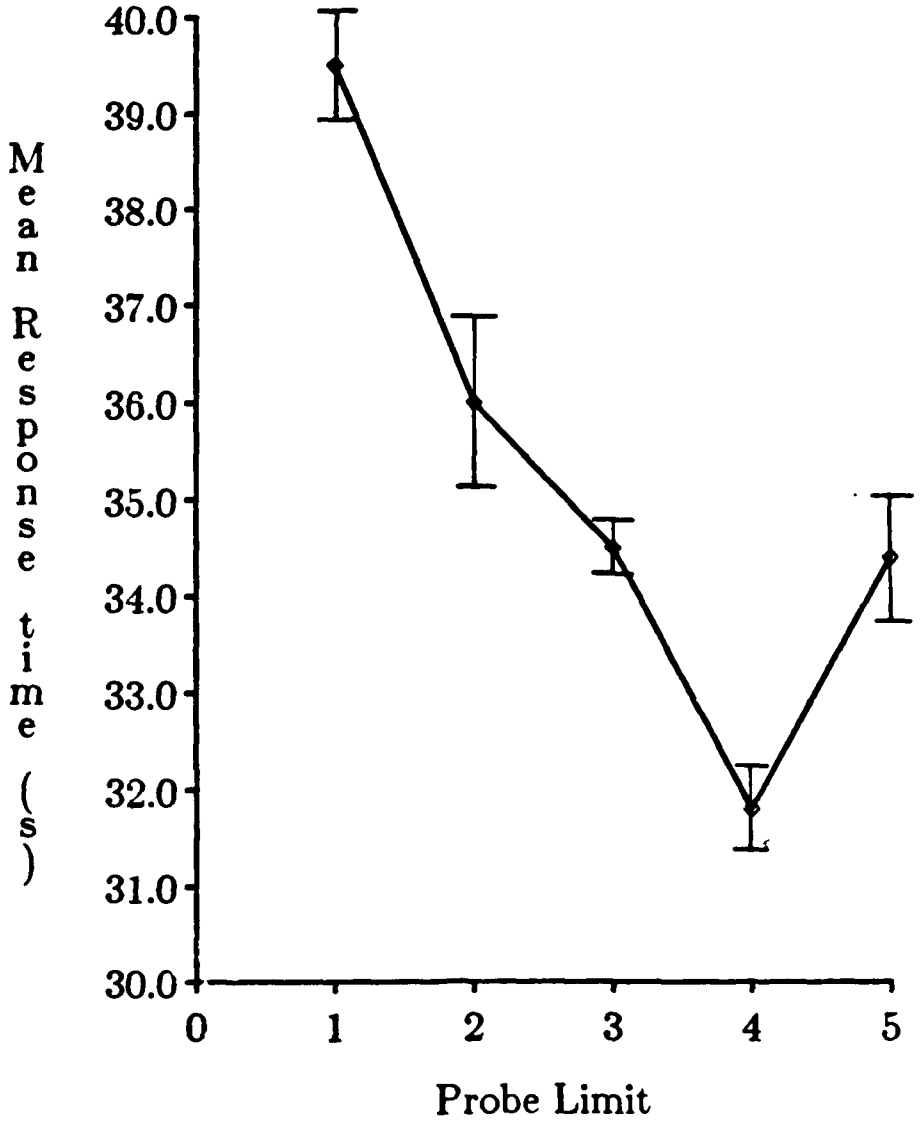

Figure 4.4. Mean process response time under various probe limits L_p (Canonical workload, LOWEST, T_l=0.8).

the performance gains can be easily explained. For the workloads used in this section, however, the hosts are similarly loaded, yet sizable reductions in response times are observed for the heavy and moderate workload cases. These gains can only be attributed to the *short-term* host load imbalances. At any particular point in time, some hosts are likely to be significantly less loaded than others, hence transferring jobs to them will reduce the overall mean job response time. The distinction between senders and receivers is not clear here; a host may be overloaded and transfer jobs out at one time, and underloaded later, when it will receive jobs from other hosts.

Table 4.8. Five hosts with heavy loads (P= 10.0 s, T_l= 1.0).

Algorithm	Resp. Time	Improv.	Std. Dev.	Improv.
NoLB	87.0 ±2.03	—	121.4	—
GLOBAL	59.4 ±0.15	31.7%	75.9	37.5%

Table 4.9. Six hosts with moderate loads (P= 10.0 s, T_l= 0.8).

Algorithm	Resp. Time	Impro.	Std. Dev.	Improv.
NoLB	49.5 ±0.27	—	72.4	—
GLOBAL	39.4 ±0.44	20.5%	57.5	20.6%

Table 4.10. Six hosts with light loads (P= 10.0 s, T_l= 0.6).

Algorithm	Resp. Time	Impro.	Std. Dev.	Improv.
NoLB	28.7 ±0.65	—	38.7	—
GLOBAL	25.2 ±0.52	12.2%	31.4	18.9%

A comparison between the response time reductions in the three cases show that the higher the system load, the greater performance improvement may be expected. This agrees with our simulation results (see Figure 3.3), and is highly desirable. Also, it should be noted that the reductions in the standard deviation of the process response times when the hosts are evenly loaded are not as large as in the long-term unbalanced case discussed in Section 4.5.

The reader may have noticed that, while six workstations were used for the moderate and light workloads, only five have been used for the heavy workload. This is because in the latter case the file server was heavily congested by file requests. In our experimental system, all the workstations get their files, and all but two of the workstations do remote paging and swapping, from a single file

server, which is also shared by other workstations, and is simply another Sun-2 workstation configured with disks. When the six workstations are active, the load on the file server becomes higher than that on the workstations, even for the moderate workload case. Under a heavy workload, the file server can be overwhelmed by file access and paging requests, with its average load index going up to 6 and over. Our experience agrees well with the results of a performance study of diskless workstations by Lazowska *et al.*, in which the authors concluded that the file server's CPU tends to be the first resource in the system to saturate [Lazowska86].

With the file server's CPU being the focus of contention, the system is no longer correctly configured, and the potential benefits of load balancing are overshadowed by the negative impact of a major I/O bottleneck. We conjecture, therefore, that greater performance gains are possible if more powerful and/or multiple file servers are provided. A load index value of 3 is considered to represent a heavy load in our workstation environment, but may be considered quite normal in compute servers or time-sharing systems. With the possibly higher loads in those types of environment, the utility of load balancing should be greater.

4.6.2. Different Mobilities

In Section 3.4.4, it was pointed out that some of the jobs have to be executed locally (immobile jobs). The effects of these jobs are studied here with measurement, and compared to the earlier simulation results. We use a slightly different definition of the immobility factor here than that in simulation: τ is defined as the percentage of CPU time consumed by the immobile jobs over all jobs. The impact of immobile jobs on the mean and variance of the job response times are depicted in Figures 4.5 and 4.6.

The different values of the immobility factor shown in the graph were obtained by changing the list of eligible jobs in the configuration file, as we can easily measure the total amount of CPU times consumed by each type of jobs, and compute their respective percentages of the total. Note that the canonical workload used in all the previous sections corresponds to an immobility factor of 0.17. As we have observed in simulation, the curves are distinctly concave upward. Even when the immobility factor is as high as 0.8 (i.e., 80% of the workload is immobile), most of the performance gains of load balancing are still retained. This seems to suggest that only a small percentage of the jobs need to be transferred among the hosts to achieve effective load balancing. For a wide range of immobility factor values and other adjustable parameters, we have observed that only less than half of the eligible jobs are actually transferred. Comparing the curve in Figure 4.5 to those in Figure 3.6 from simulation, we notice a striking similarity in their shapes. Since the definition of the immobility factor τ that we use here includes all jobs, as opposed to only the eligible jobs as

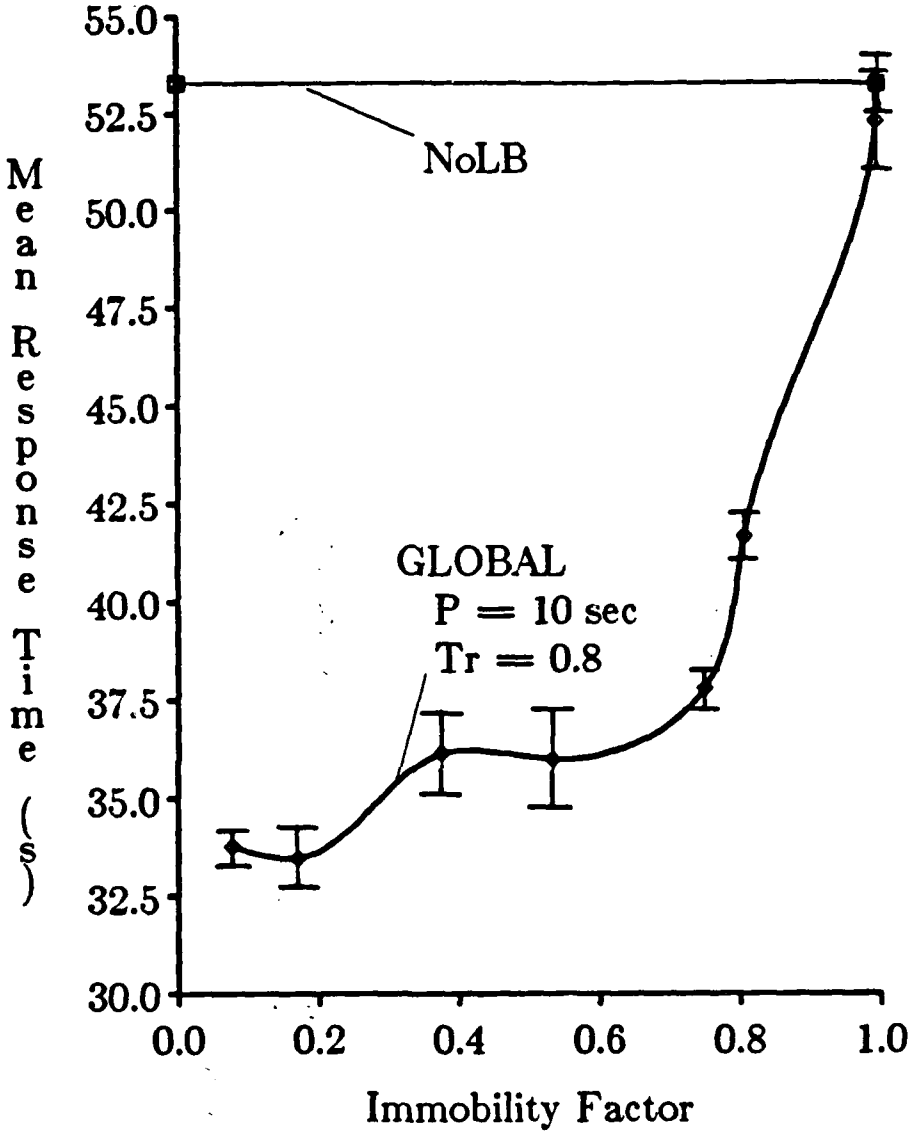

Figure 4.5. The influence of immobile jobs: mean response time vs. immobility factor (Canonical workload, GLOBAL).

the one used in simulation, the point at which performance starts to degrade significantly corresponds to a higher value of τ.

4.7. Effects on Individual Hosts and Job Types

In the above two sections, we have studied the influences of the two major factors, namely, the algorithms and the workloads, on load balancing performance. We go into more detailed studies in this section by examining the impact of load balancing on the loading and performance of the individual hosts and on the response times of each type of job.

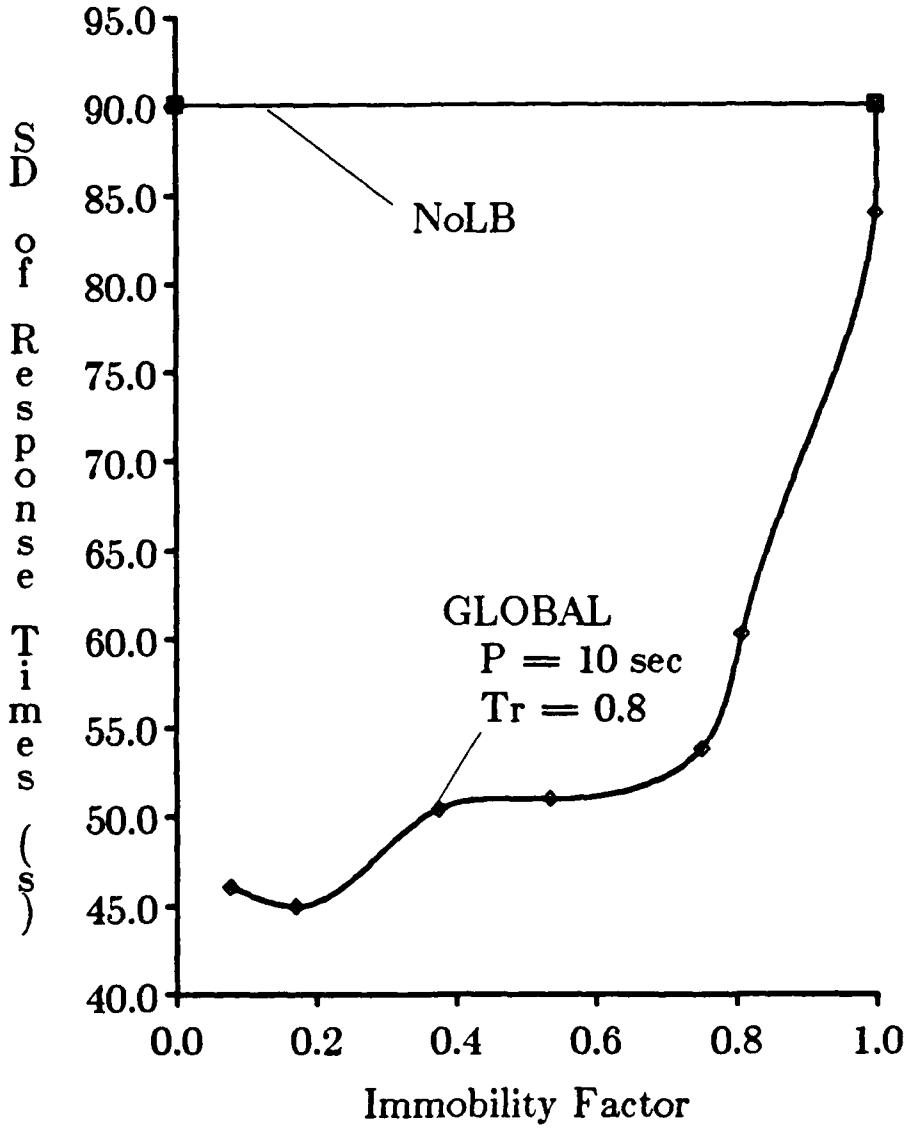

Figure 4.6. The influence of immobile jobs: standard deviation of response times vs. immobility factor (Canonical workload, GLOBAL).

4.7.1. Effects on Individual Hosts

Although it is now clear that load balancing can improve system-wide performance, its impact on the loading of individual hosts is equally important, especially in a workstation environment. Figure 4.7 shows the average load index value of each host throughout a run, and with different values of the immobility factor τ. We see a significant reduction in the loads of all the hosts except those that were originally very lightly loaded. This is a confirmation of the reduction in the average response times we observed, and is in agreement with Little's result, which states that the mean queue length and the mean

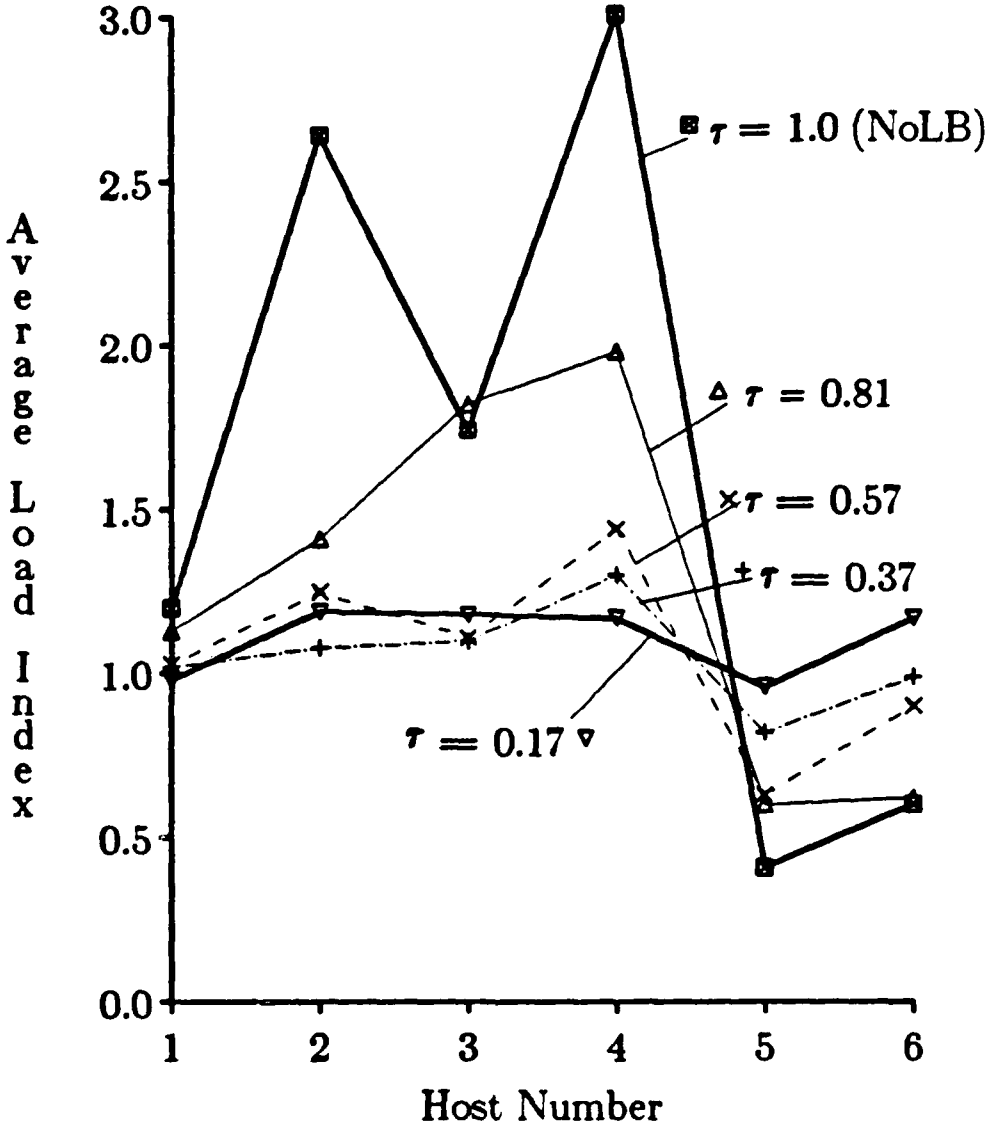

Figure 4.7. Loads on hosts, with various immobility factors (Canonical workload, GLOBAL, P= 10.0 s, T_l= 0.8).

queueing time has a linear relationship in any queueing system. We also notice a strong equalization of the hosts' loads: as the immobility factor goes from 1.0 down to 0.17, the hosts' loads are compressed into a narrow range. Thus the term "load balancing" is truly appropriate in our case, even though none of the algorithms we studied takes equalizing loads as its explicit objective. The similarity between Figures 4.7 and 3.10 is very easy to notice.

The fact that the loads of the hosts tend to become balanced on the average does not necessarily mean that they are balanced during shorter intervals, which,

however, would be highly desirable. Indeed, this is shown again not to be the case by Figure 4.8, where the 20 second averaged sum of CPU, I/O, and memory queue lengths (as a load index; see Table 4.5) is plotted as a function of the time during a run.

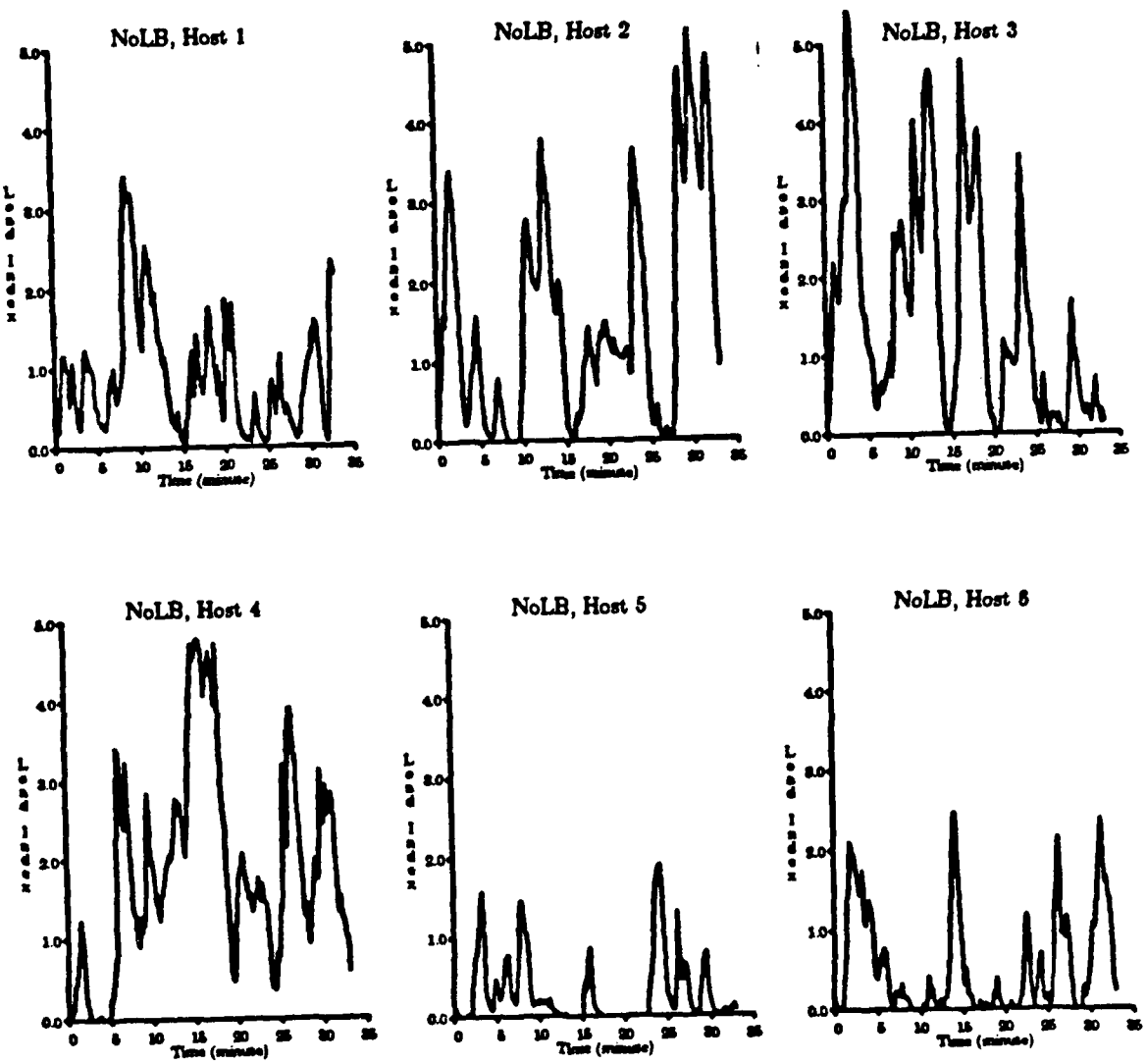

Figure 4.8a. The load on each host as a function of time, NoLB.

Several comparisons may be made using the plots. Comparing the loads of the hosts without load balancing, we see significant differences in loads. These

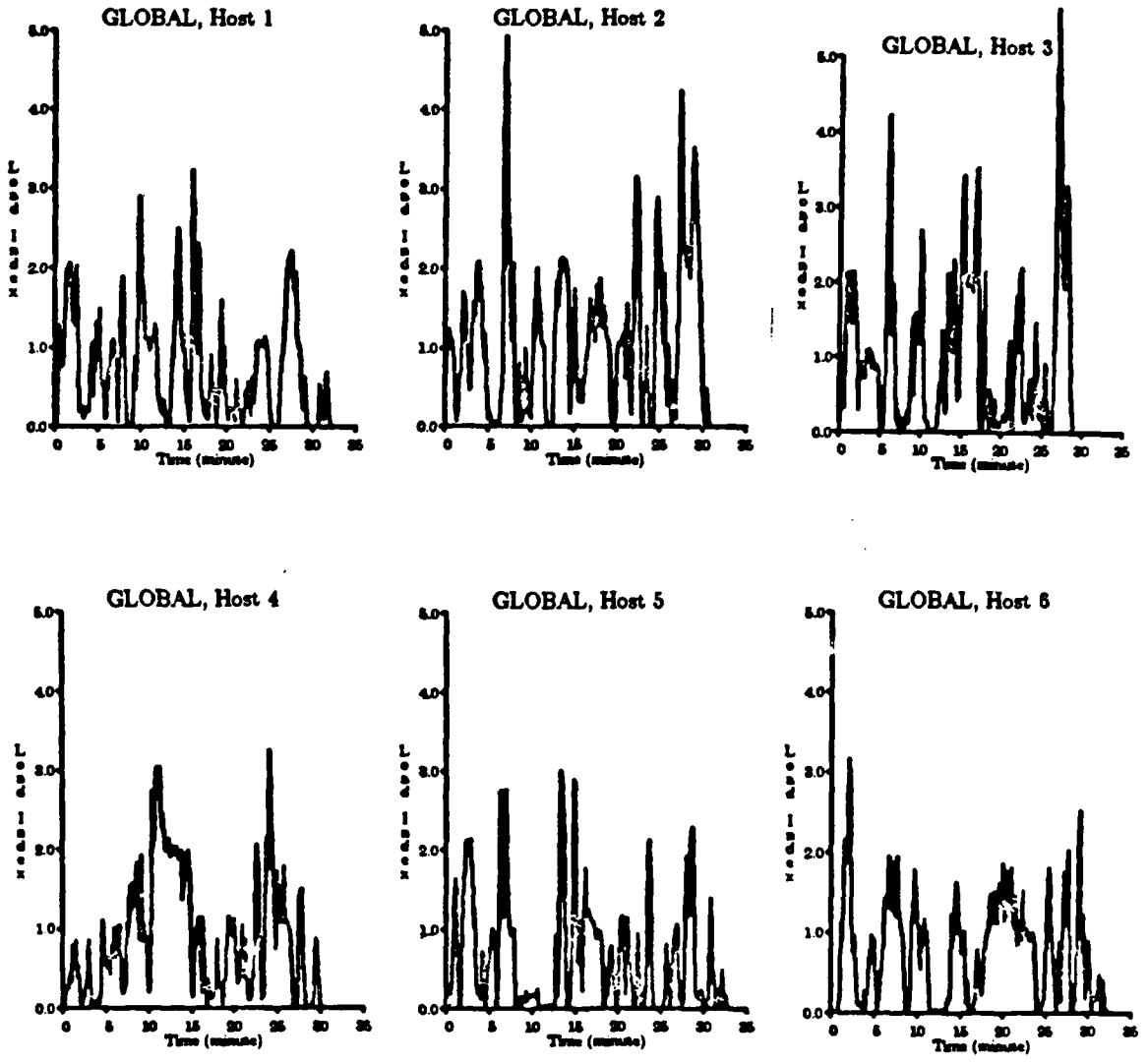

Figure 4.8b. The load on each host as a function of time, GLOBAL.

differences are substantially reduced by load balancing. However, there still exist load fluctuations in each host. Actually, the fluctuations are stronger than those we observed in simulation (see Figure 3.11). Our load balancer operates at the job level, and several processes may be created by a single job. As long as those processes are treated as an inseparable group, temporal fluctuations in load seem unavoidable. Since smoothing the hosts' load over time is highly desirable, we conclude that load balancing at the job level using initial placement only has

the drawback of not being able to eliminate temporal fluctuations. On the other hand, it is questionable whether the performance gains due to further reductions in temporal load fluctuations provided by load balancing at a finer granularity would more than offset the additional communication and computation overhead. More research is called for here.

4.7.2. Effects on Each Type of Jobs

The conjecture could be made that, while the mobile jobs will generally benefit from load balancing, the immobile jobs will not benefit much, or not at all. Our measurements contradict this conjecture. Table 4.11 lists the mean response times of each type of jobs executed during the runs with and without load balancing. All times are in seconds, and the percentage improvements are provided (in parenthesis) following the response times for the load balancing case. As shown in Table 4.7, the average response time of all jobs changed from 53.3 s for NoLB to 32.6 s for GLOBAL, with an improvement of 38.9%.

The average response times of *all* types of jobs are reduced, and, with only a few exceptions (*cp, date* and *finger*), the reductions are uniform across the board. There is no clear difference in improvements between different classes of jobs, big or small, mobile or immobile. While the response time of a job to be transferred will improve because it will be executed on a more lightly loaded host, those of the jobs already running on the initial host will also improve because they will not have to compete with the newcomer.

4.8. Summary

In this chapter, we described the design and implementation of a load balancer for a loosely-coupled distributed system, and presented some of the results of a large number of measurement experiments performed on the system. On the basis of our findings, we believe that transparent, flexible load balancing at the job level can be achieved at low cost, and without modifying either the system kernel or any of the existing application programs. Our design emphasizes a clear separation between the mechanism and the policies for load balancing, thereby allowing the particular load balancing algorithm, along with its adjustable parameters (e.g., the load exchange period, the local load threshold, and the probe limit) to be dynamically changed in response to changing system load conditions.

Measurements show that load balancing can indeed significantly reduce the mean process response time, and that the corresponding reduction in the standard deviation of process response times is even greater. Furthermore, the improvements are largely uniform over all classes of jobs, big or small, mobile or immobile, and most of the improvements can still be retained when up to 80% of the workload cannot be transferred between the hosts. While we observed

Table 4.11. Average response time of each command type with and without LB (Canonical workload, GLOBAL with T_l= 0.8, P= 5 s).

cmd	elig.	count	NoLB	LB	cmd	elig.	count	NoLB	LB
cat	N	33	5.19	3.53 (32.0%)	*ls*	N	53	52.7	30.3 (42.5%)
cc	Y	54	89.1	53.8 (39.6%)	*man*	Y	8	20.2	6.78 (66.4%)
cp	N	3	2.34	2.30 (1.7%)	*mv*	N	2	3.61	1.72 (52.4%)
date	N	22	1.81	1.46 (19.3%)	*nroff*	Y	17	181	102 (43.7%)
df	N	9	6.22	3.61 (42.0%)	*ps*	N	23	22.5	14.1 (37.3%)
ditroff	Y	7	324	194 (40.1%)	*pwd*	N	18	4.26	3.02 (29.1%)
du	N	6	82.6	55.1 (33.3%)	*rm*	N	0	-	-
egrep	Y	7	22.1	6.07 (72.5%)	*sort*	N	30	105	66.8 (36.4%)
eqn	Y	5	103	64.2 (37.8%)	*spell*	Y	45	117	73.6 (37.1%)
fgrep	Y	10	19.2	11.9 (38.0%)	*tbl*	Y	2	109	55.9 (48.7%)
finger	N	25	92.6	80.4 (13.2%)	*troff*	Y	12	110	65.8 (40.2%)
grep	Y	3	12.1	6.56 (45.8%)	*uptime*	N	34	7.85	4.18 (46.8%)
grn	Y	7	277	158 (43.0%)	*users*	N	4	7.08	3.20 (54.8%)
lint	Y	24	78.6	42.0 (46.6%)	*wc*	N	15	12.3	5.13 (58.3%)
lpq	N	12	29.8	15.2 (49.0%)	*who*	N	11	4.78	2.29 (52.1%)

that load balancing has strong equalization effects on the individual hosts' loads over the entire measurement runs, there still exist temporal fluctuations in host loads. We attribute this drawback to the fact that several processes may be created by a single job, and suggest that load balancing at a finer granularity be studied to see whether this conjecture is correct, and whether such fluctuations can be advantageously reduced.

Five load balancing algorithms were studied that used different methods to distribute load information and to perform job placement. We found that the algorithms using periodic load exchanges and those acquiring such information on demand provide comparable performances. For the former class of

algorithms, the use of a central agent to collect and distribute load information reduces the computation and communication overhead, and hence provides better performance. The centralized algorithms are also better suited for adaptive load balancing, in which the algorithm and/or its parameters may be changed dynamically. On the other hand, distributed algorithms such as LOWEST generally impose lower overhead, scale better, and are more reliable. We also found that the performance of load balancing is, to various degrees, sensitive to the algorithms' parameter values.

As well as load balancing algorithms and their parameters, workloads also have a strong impact on performance. Generally speaking, the higher the load, and the greater the imbalances in the hosts' loads (both long-and short-term), the greater the performance improvements that may be expected. Short-term imbalances can be as profitably exploited as long-term imbalances, as demonstrated by the performance gains when all the hosts are subjected to similar levels of loads.

In Sections 4.5, 4.6, and 4.7, we compared each aspect of our measurement results to those in Section 3.4 from simulation, and found good agreements in almost all cases. As the two independent studies use different approaches, workloads, and types of system, the results from measurements serve as strong support to our simulation approach and its findings.

Chapter 5

Load Balancing in Other Environments

5.1. Overview

The results of the simulation experiments reported in Chapter 3 and those from measurements in Chapter 4 agree well in many aspects. In this chapter, we drive the same type of simulator used in Chapter 3 with trace data from two other computing environments, Bell Communications Laboratory and Lawrence Berkeley Laboratory, to further test the generality of our findings. A comparative study of the three workloads and their corresponding simulation results will certainly extend and enhance our understanding of load balancing.

In the next section, we characterize the workloads from Berkeley, Bell, and LBL by their job arrival, and CPU and I/O consumption patterns. While interesting in itself, workload characterization also provides a basis for the explanation of the performance differences among the systems. In Sections 5.3 and 5.4, simulation experiments using the Bell data and the LBL data are discussed. Section 5.5 is a brief summary.

5.2. Workload Characterization

Our purpose for using multiple sets of traces in simulation is to identify those observations common to all the systems, which are likely to be general properties of load balancing, rather than being peculiar to some system. Consequently, the workloads should be chosen from environments as diverse as possible. Unfortunately, collecting traces from computer systems often involves a substantial amount of effort.

Besides the Berkeley trace, we managed to collect or obtain traces with the same format from two other environments. Like the Berkeley data, the Bell data is from VAX-11/780 machines running Berkeley UNIX, but these machines support research computing over a wider spectrum of topics, and in an industrial setting†.

---

† Leland and Ott collected the data, and used it in their simulation work [Leland86].

The LBL data is from a cluster of five VAX-11/8650 machines running the VMS operating system and sharing disks via a high speed bus. The 8650 processor is generally regarded as 4-5 times more powerful than a 780 processor, and, also because of the large main memory of several hundred MB in each processor, the LBL environment may be regarded as a bank of compute servers. The workload is a combination of daily computing and large-scale scientific computation, simulation, and graphics applications in disciplines ranging from physics, to chemistry, to astronomy. Measurements have been collected simultaneously from all of the five machines. Simulated systems with size greater than five are driven by traces from several days. It has been observed that, due to the very large main memory size, the contention for the shared bus and the disk drives were light, and the CPUs were definitely the system bottlenecks‡.

In Tables 5.1, 5.2, and 5.3, some basic statistics are provided for the three environments. For ease of comparison, the Berkeley data are repeated from Chapter 3. In all cases, the job resource demand distributions are far from exponential, as pointed out in Chapter 2. The ratio between the standard deviation of job CPU demands and their mean ranges from 6.1 in the Berkeley data to 36 in the LBL data. The same ratio for file I/O ranges from 4.5 to 9. Consistently, the Bell data demonstrates greater skewness in resource demand distributions than the Berkeley data, and the LBL data an even greater one. In the former case, this is caused by the substantial number of big jobs commonly seen in industrial environments. In the latter case, the high skewness results not only from the very heavy jobs, but also from the greater power of the CPUs --- the daily computing jobs take very little time to finish.

To gain more detailed understanding of the job resource demand patterns, we generated three groups of distributions from the three sets of traces. Figure 5.1 shows the cumulative distributions of the number of jobs with CPU demands less than or equal to x from measurements in the three environments, compared to the corresponding curves for an exponential distribution with the same mean. In all three environments, there is a substantially larger portion of jobs with little CPU demand compared to the exponential distribution, complemented by a small number of CPU intensive jobs. For example, 60-83% of the jobs consume less than half a second of CPU time, and only 3-7% of the jobs requires more than 4 seconds, in all the traces. All three distributions are significantly different from their exponential models, and especially the LBL data. This is consistent with our observations earlier using their mean and standard deviation.

---

‡ An informal experiment consisting of replacing magnetic disks with high speed electronic disks yields little improvement in the response times of benchmark programs, thus supporting our observation [Beals87].

A more relevant measure than the proportion of jobs is probably the percentage of CPU resources consumed by small and big jobs. Figure 5.2 provides a direct assessment.

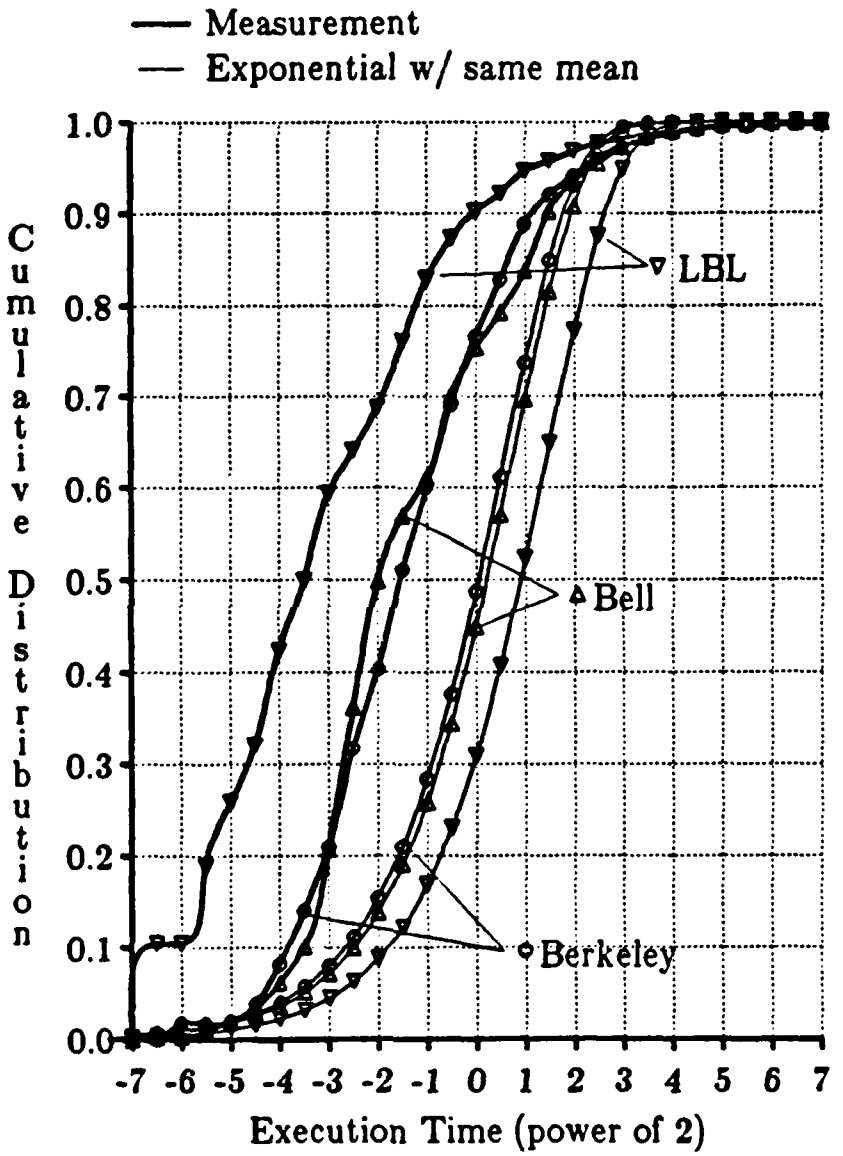

Figure 5.1. Distributions of jobs by their CPU demands.

85-95% of the total CPU time is consumed by jobs requiring more than 1 second of CPU time, or 10-25% of all the jobs. Obviously, considering only these jobs will provide effective load balancing, without causing excess overhead.

Similar to job CPU demands, the three distributions of the number of file I/O operations are also highly skewed (Figure 5.3).

Table 5.1. Basic Statistics of the Berkeley Data.

Total duration:	196 hrs (49 sessions of 4 hrs each)
Total number of jobs:	297,595
Job inter-arrival time:	mean= 2.371 s, SD= 6.270 s
Job execution time:	mean= 1.492 s, SD= 19.14 s
Number of file I/Os per job:	mean=18.23, SD= 81.43
Average CPU utilization:	62.9%
Average response time (NoLB):	5.38 s
Average CPU queue length (NoLB):	2.03

Table 5.2. Basic Statistics of the Bell Data.

Total duration:	1008 hrs (42 sessions of 24 hrs each)
Total number of jobs:	1,168,579
Job inter-arrival time:	mean= 3.105 s, SD= 11.70 s
Job execution time:	mean= 1.675 s, SD= 51.88 s
Number of file I/Os per job:	mean=23.87, SD= 147.9
Average CPU utilization:	53.9%
Average response time (NoLB):	7.89 s
Average CPU queue length (NoLB):	2.28

Table 5.3. Basic Statistics of the LBL Data.

Total duration:	960 hrs (40 sessions of 24 hrs each)
Total number of jobs:	592,661
Job inter-arrival time:	mean= 5.831 s, SD= 73.91 s
Job execution time:	mean= 2.702 s, SD= 98.33 s
Number of file I/Os per job:	mean=58.83, SD= 527.9
Average CPU utilization:	46.3%
Average response time (NoLB):	6.56 s
Average CPU queue length (NoLB):	0.81

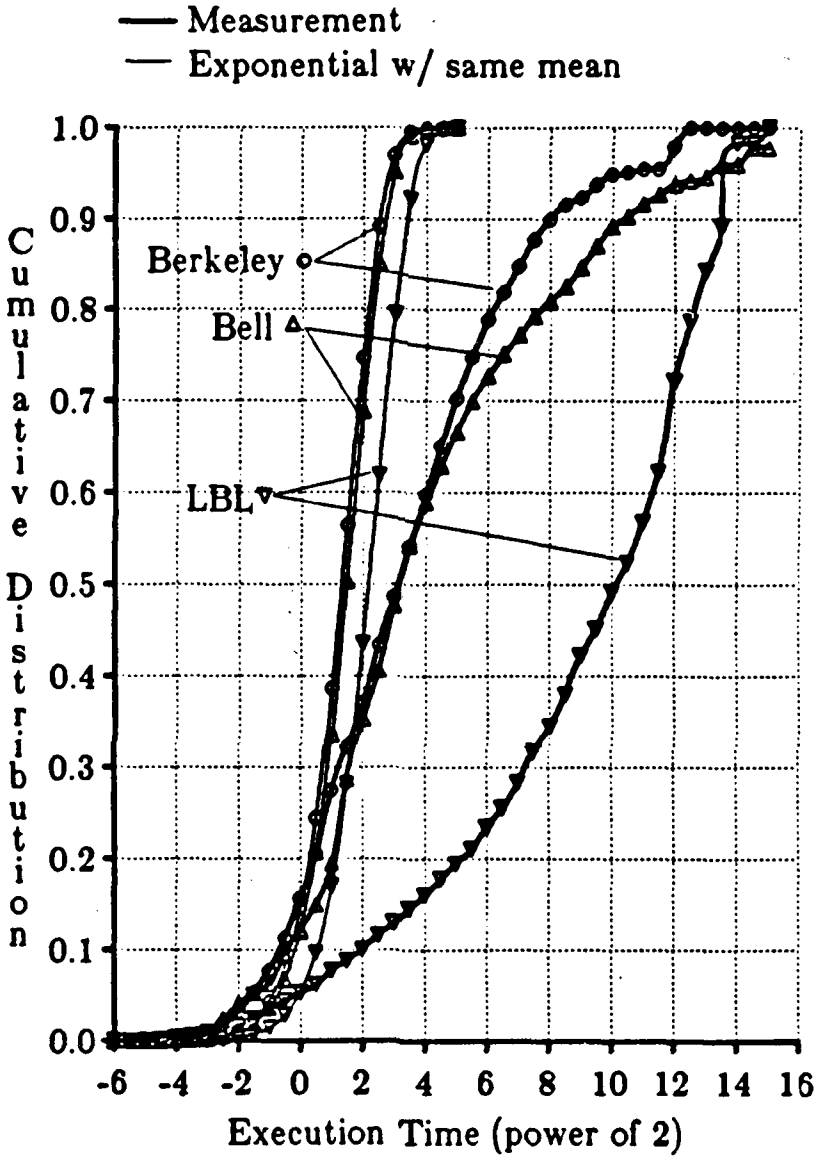

Figure 5.2. Cumulative distributions of CPU time consumed by jobs of different sizes.

5.3. Simulations Driven by the Bell Traces

With an understanding of the similarities and differences in the three sets of trace data, we are ready to discuss simulation experiments with the *Bell* and LBL traces in this and the next section. For the Bell trace, the same system model as that for Berkeley was used. A large set of experiments have been performed, and the results were found to be in excellent agreement with those from the Berkeley trace. To avoid unnecessary repetition, however, we choose to present only some of the more interesting and possibly controversial results here. The structure of the following subsections parallels those in Section 3.4, with

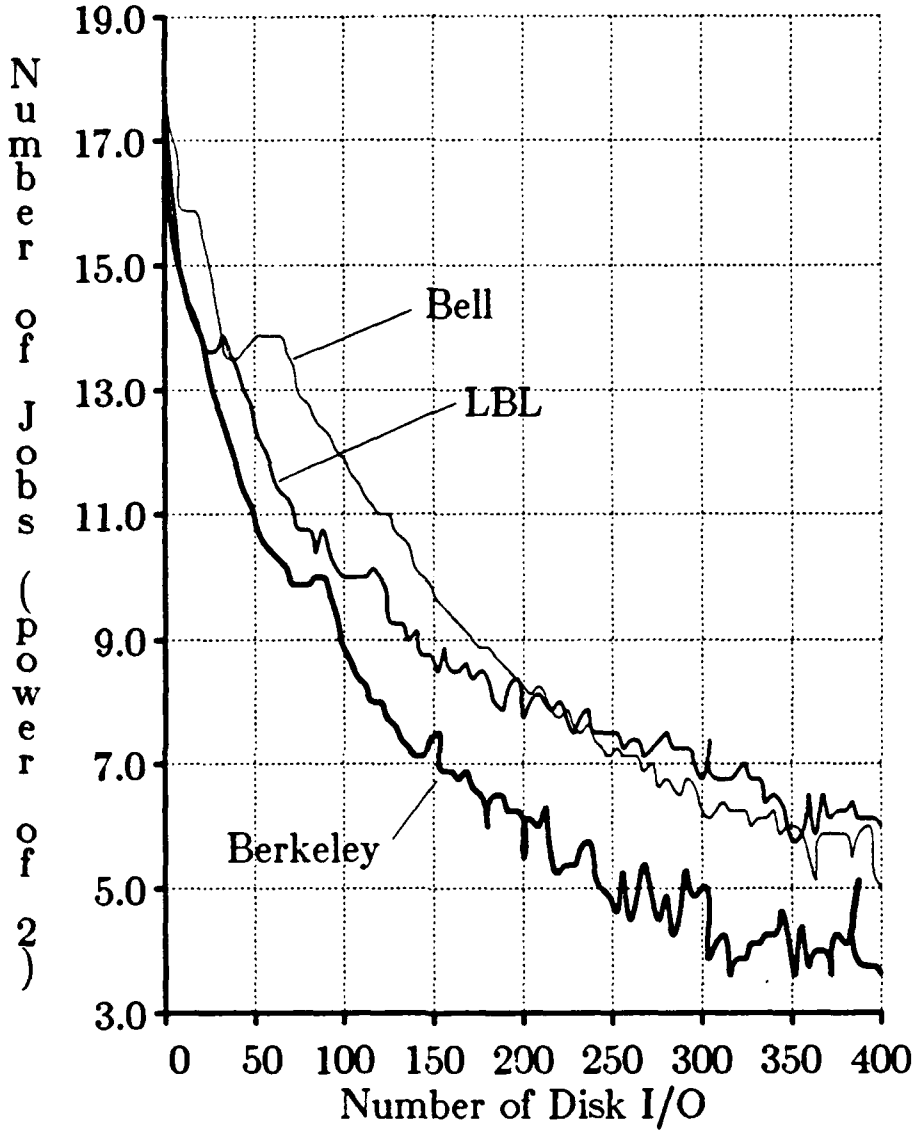

Figure 5.3. Distributions of jobs by their file I/O.

some subsections omitted.

5.3.1. Comparison of the Algorithms

Figure 5.4 shows the best achievable performances of the seven algorithms, together with the three boundary cases for systems with 7, 14, 28, and 42 hosts and under comparable levels of load. Since the networking environments are similar for Berkeley and Bell (Ethernet type local area network), we used the same overhead assumptions for the Bell systems as for the Berkeley ones. The curious reader can easily verify the similarities between Figures 5.4 and 3.2.

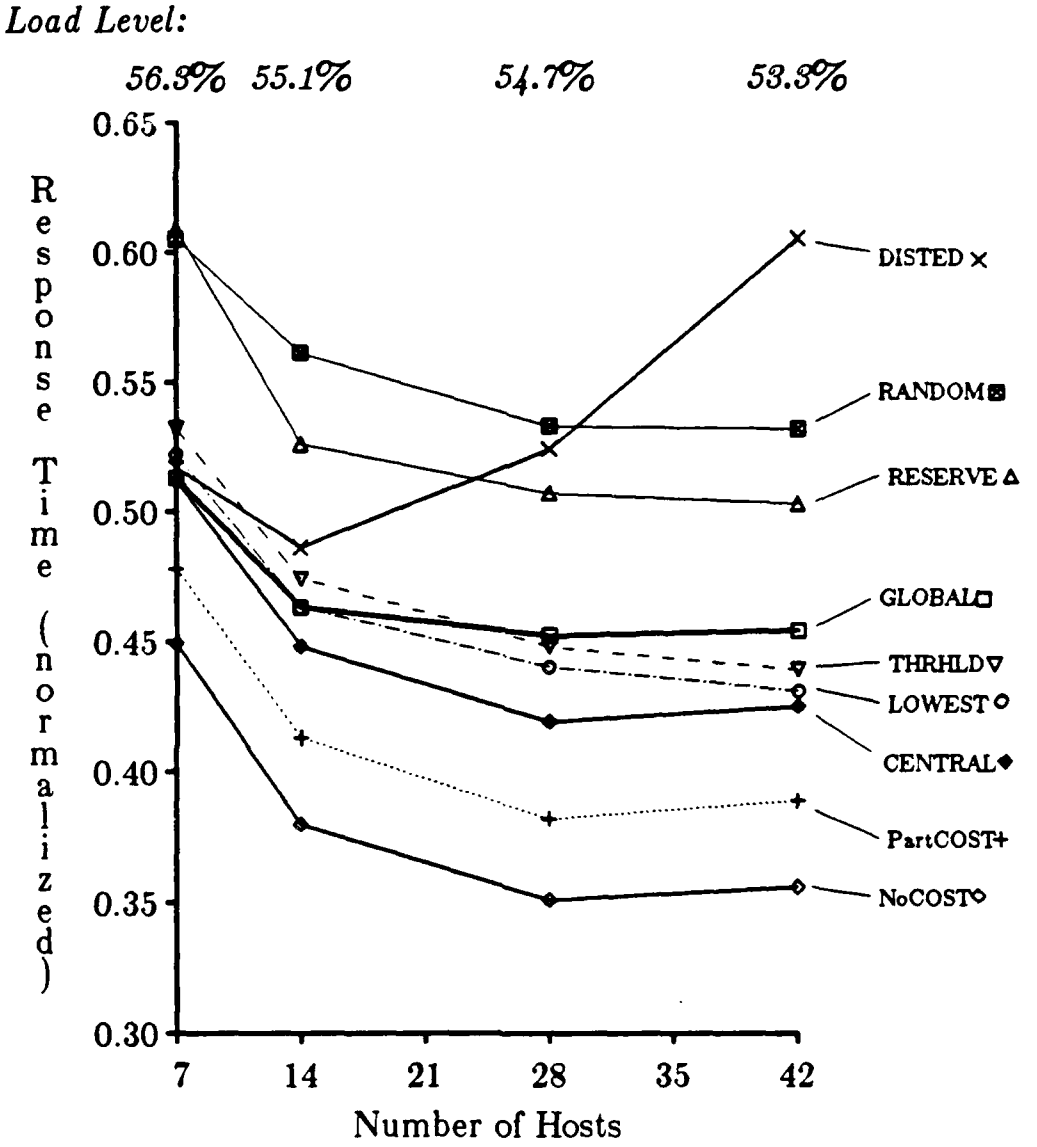

Figure 5.4. Average response times with different system sizes (Bell trace; normalized against the NoLB case).

The amount of performance improvement achievable with the Bell data is approximately 5 percent more than that with the Berkeley data, although the overall system load level, measured by the average CPU utilization, is lower. This appears to contradict our earlier observation that improvement increases with load level. A more careful examination, however, shows that the level of resource contention in the Bell systems during the normal working hours is actually higher than that in the Berkeley system. Due to the 24 hour duration of the session period, however, the overall load level is lower. The improvement, on the other hand, is greater, as a large proportion of the jobs are executed during

the heavy congestion periods.

The relative rankings of most of the algorithms in Figure 5.4 are the same as in Figure 3.2. GLOBAL, THRHLD, LOWEST, and CENTRAL are still the best performing algorithms, with improvements close to each other and not far from that of NoCOST. RANDOM performs quite well considering its simplicity. while RESERVE and DISTED yield performances significantly worse than those of the best performing ones. The sharply rising curve of DISTED is especially noticeable.

The scaling behavior of the algorithms again shows significant but limited economies of scale, as in Figure 3.2. Even the points where the curves start to flatten are very close --- around 28 hosts. Two of the three centralized algorithms, GLOBAL and CENTRAL, still demonstrate good scaling capability.

A shortcoming in the Berkeley trace is its relatively short 4-hour session duration. Very long jobs may not be reflected, and possible temporal correlation between the hosts' loads† may be lost. This problem is reduced in the Bell and LBL traces, where all jobs started during the 24 hour period are included, and, since all sessions start and finish at midnight, some recurrent system loading patterns are captured. The similarities in the results for the Berkeley and the Bell data seem to suggest that no significant error is introduced by the shorter sessions of the Berkeley systems.

5.3.2. Effect of Immobile Jobs

We repeated the experiment in Section 3.4.4 using the Bell trace, and the performance with different values of the immobility factor is shown in Figure 5.5. The striking similarities among the three corresponding plots in Chapters 3, 4, and 5 strongly suggest that the insensitivity of performance to the changes in the fraction of immobile jobs is a fundamental property of load balancing. This conclusion is also supported by the LBL data.

5.3.3. Impact on Individual Hosts

Again, experiments parallel to those in Section 3.4.6 were performed with a system of fourteen hosts, so that the utilizations, the means and standard deviations of the job response times, and the average CPU queue lengths of the hosts, with and without load balancing could be compared. The results are shown in Figures 5.6, 5.7, 5.8, and 5.9.

With the host CPU utilization with load balancing ranging from 45% to 70%, the average response time of every host is improved (Figure 5.7). Host 4

† For example, the loads of all hosts may increase shortly before the lunch hour, as people submit long jobs before leaving.

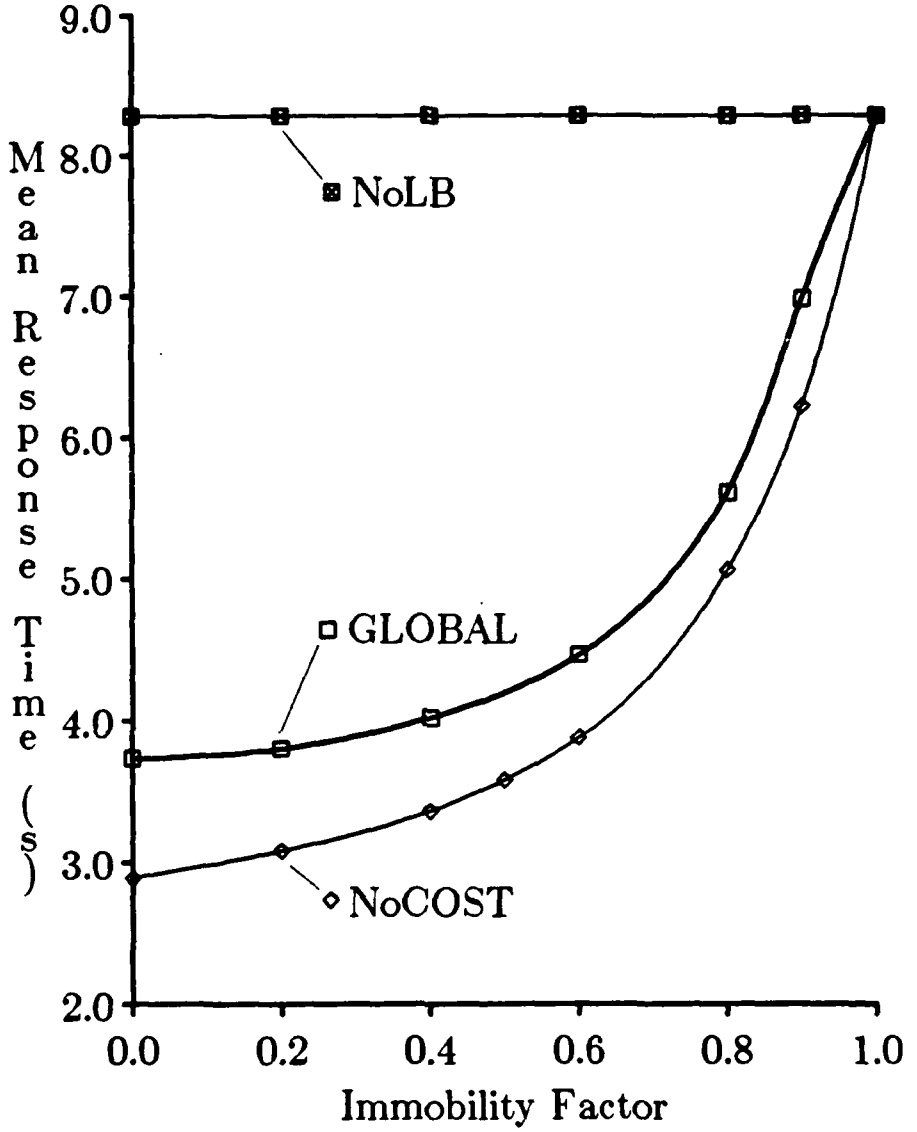

Figure 5.5. Effect of immobile jobs (Bell trace; T_{CPU}=1.0 s).

experiences only a small improvement, although its utilization is relatively high. Looking at its load more closely, we find that a number of very long jobs were executed on it sequentially, and that the competition for CPU was low (see its average CPU queue length in Figure 5.9). Load balancing at the job level cannot, and should not help much in this situation. This example also reveals the inadequacy of the average CPU utilization as the only measure of load level. The improvement in the standard deviation of job response times is less substantial than in the Berkeley trace simulations and in the measurement results. The comparison of the average CPU queue lengths with and without load balancing yields no surprise: the values are reduced significantly, and fall into a very

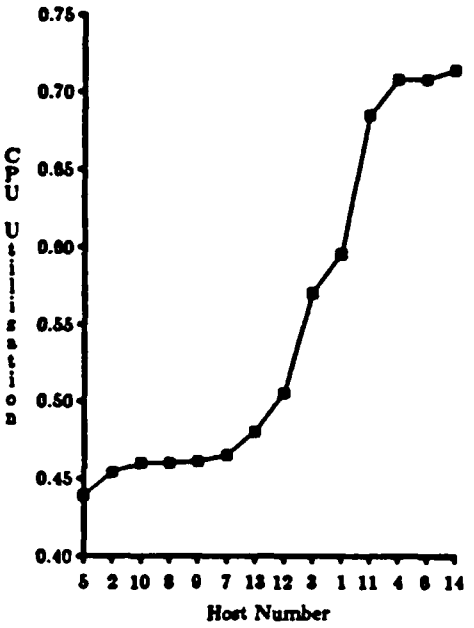

Figure 5.6. Host CPU Utilizations of a 14 Host System (sorted for NoLB; average utilization with NoLB: 55.1%).

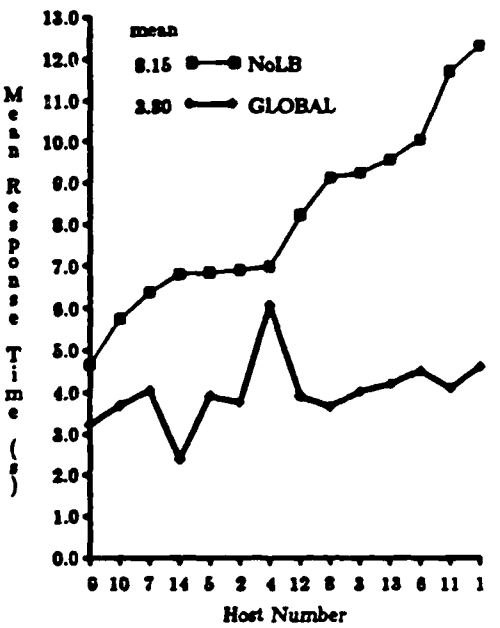

Figure 5.7. Mean response times of individual hosts (sorted for NoLB; average utilization with NoLB: 55.1%).

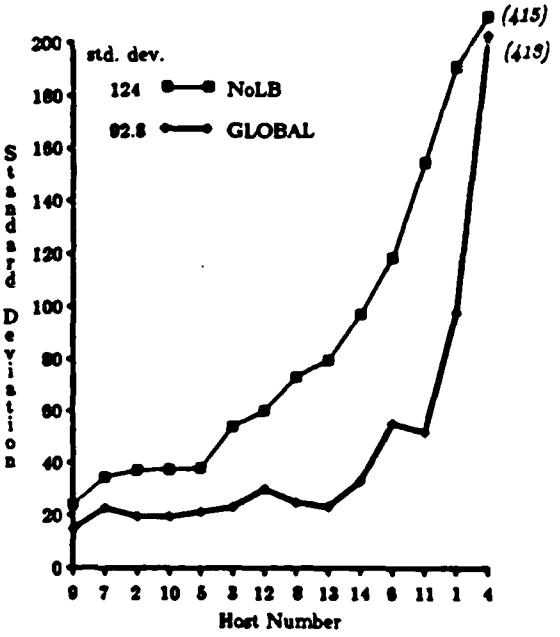

Figure 5.8. Standard deviation of response times of individual hosts (sorted for NoLB; average utilization with NoLB: 55.1%).

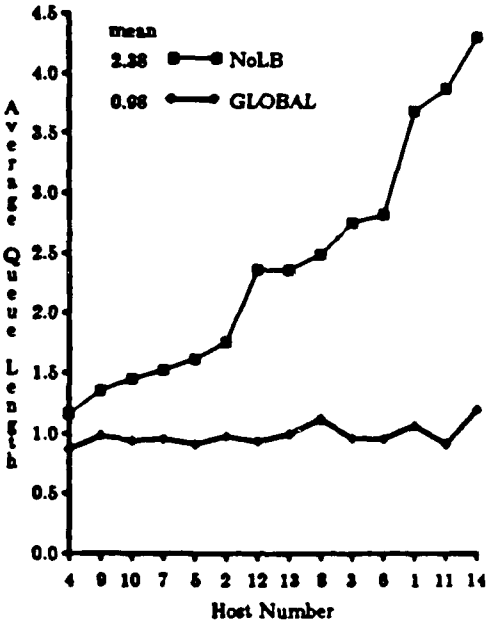

Figure 5.9. Average queue lengths of individual hosts (sorted for NoLB; average utilization with NoLB: 55.1%).

narrow range. An examination of the time behaviors of the host loads show reduced, but still existent fluctuations.

5.3.4. Impact on Each Job Class

Just as we did in Section 3.4.7, the average response times of different kinds of jobs were computed from simulations of a 28-host Bell system. The results are shown in Figures 5.10 and 5.11. (In Figure 5.11, E represents the amount of CPU time a job consumes.) The only difference between these plots and Figures 3.14 and 3.15 we can notice is that the performance difference between the Remote and Local jobs is somewhat larger in the Bell system. The same reasons provided in Section 3.4.7 can be used to explain that difference here.

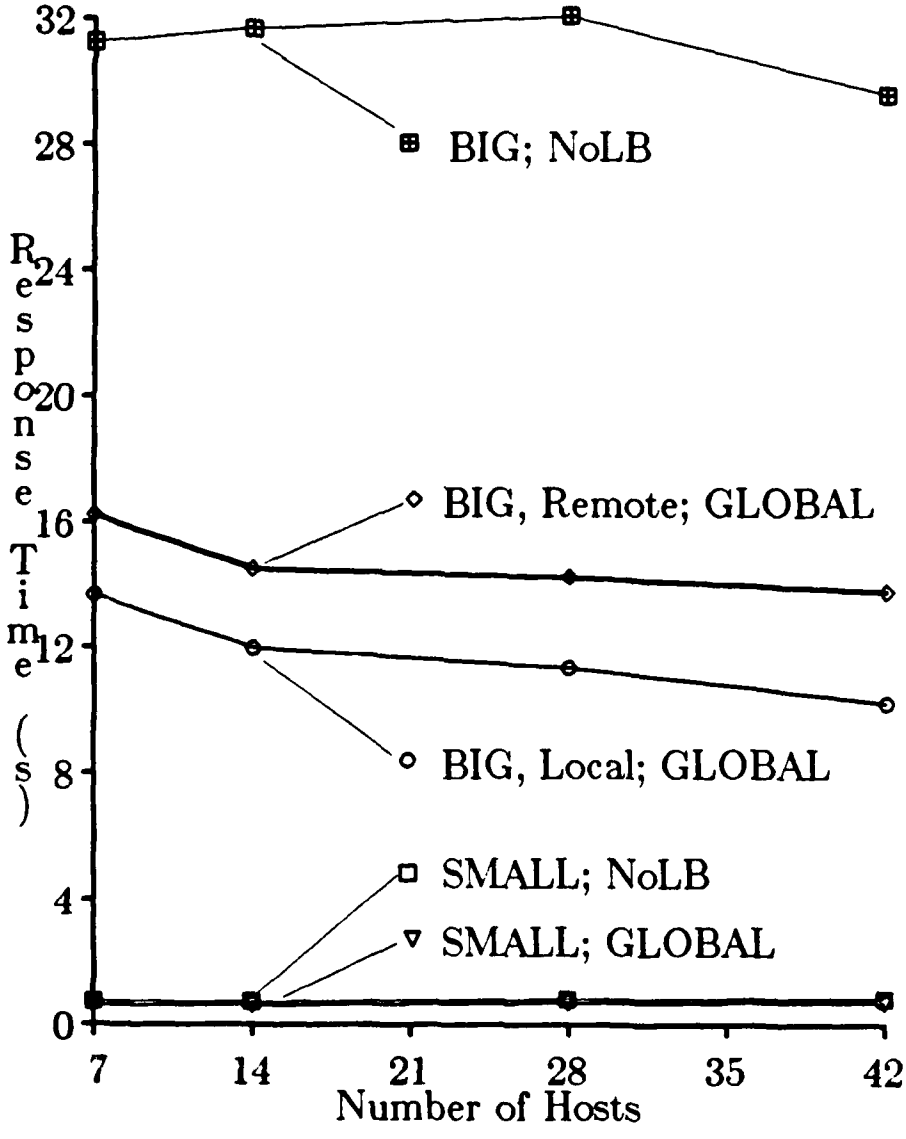

Figure 5.10. Average response times for several classes of jobs (Bell trace; 28 hosts; GLOBAL, T_l=1.0, T_{CPU}=1.0s; utilizations same as in Figure 5.4).

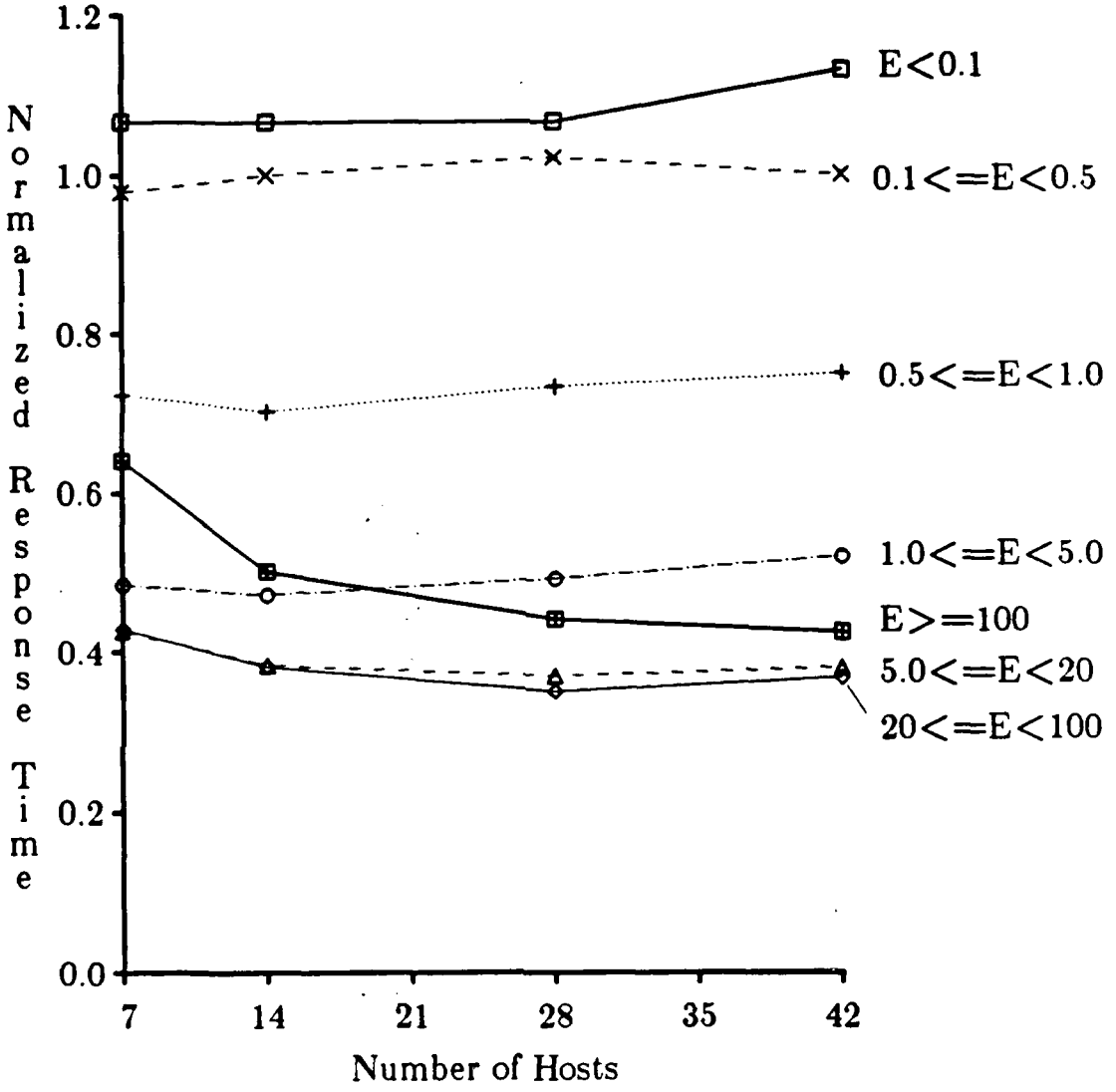

Figure 5.11. Average response times for several sizes of jobs (Bell trace; 28 hosts; GLOBAL, T_l=1.0, T_{CPU}=1.0s; utilizations same as in Figure 5.4).

5.4. Simulations Driven by the LBL Traces

The architecture of the system at Lawrence Berkeley Laboratory is quite different from those of the Berkeley and Bell systems. In a Vax cluster, a number of CPU-memory pairs are connected by a high speed bus to a pool of disk drives. Such a configuration is sometimes referred to as a "closely-coupled" distributed system, as it represents an intermediate solution between computers connected by networks (loosely-coupled), and processors sharing main memory (tightly-coupled). Due to bus contention, closely-coupled systems usually have

limited scalability. Since we are not interested in a particular networking technology, we just assume that the hosts are connected by a fast communication medium, and ignore the contention problem on the network.

To simulate the LBL systems, the same simulator structure is used. However, the message and job transfer overhead is assumed to be less than that in loosely-couple systems with VAX-11/780 type of machines, because the machines at LBL are much more powerful, and communication is faster. Ideally, a load balancer should be implemented, and the measured overhead used in the simulations. Instead, we decided to just use rough estimates. As will be seen below, as long as the costs are significantly lower than those assumed in our previous simulations, the actual overhead values do not affect the results much. We assume that computing the local load and sending its value out costs 10 milliseconds, and receiving a load value and storing it costs 5 milliseconds. Transferring a job is assumed to consume 50 milliseconds of CPU time on each side, and a 100 milliseconds total delay to the job.

Figure 5.12 shows the mean response times given by the various algorithms in systems of 5, 10, 20, and 40 hosts. The amount of improvement achieved is much less than those in the Berkeley and the Bell systems. This is consistent with our earlier observations, as the load level in the LBL systems is significant lower. With a CPU utilization of 46%, and average queue length of 0.8, load balancing is not much needed. Since the hosts are several times more powerful than those in the Berkeley and the Bell systems, we observe a large number of very small jobs. On the other hand, there are big simulation and scientific computation jobs that run for hours, usually in sequence. Just as in the case of Host 4 in Section 5.3.3, load balancing cannot help much in this case.

Due to the low cost of overhead and the reduced room for performance improvement, the differences between the performances of the algorithms are greatly diminished (except for RANDOM). This suggests that, in a system with fast interprocessor communication, reducing load balancing overhead is not as important. This observation is stronger here than in Figures 3.7 and 3.8, where the reductions (and increases) of message and job transfer costs are considered *separately*. The way the load balancing algorithms scale is still the same as in other systems, although less significant now.

Our observations of performance as a function of the immobility factor, of individual hosts, and of different types of jobs using the LBL data yield results similar to those provided by the Berkeley and Bell traces.

5.5. Summary

The studies of load balancing performance using trace data from Berkeley performed in Chapter 3 were repeated for the Bell and LBL systems. Although the results varied due to the differences in the systems and their workloads, the

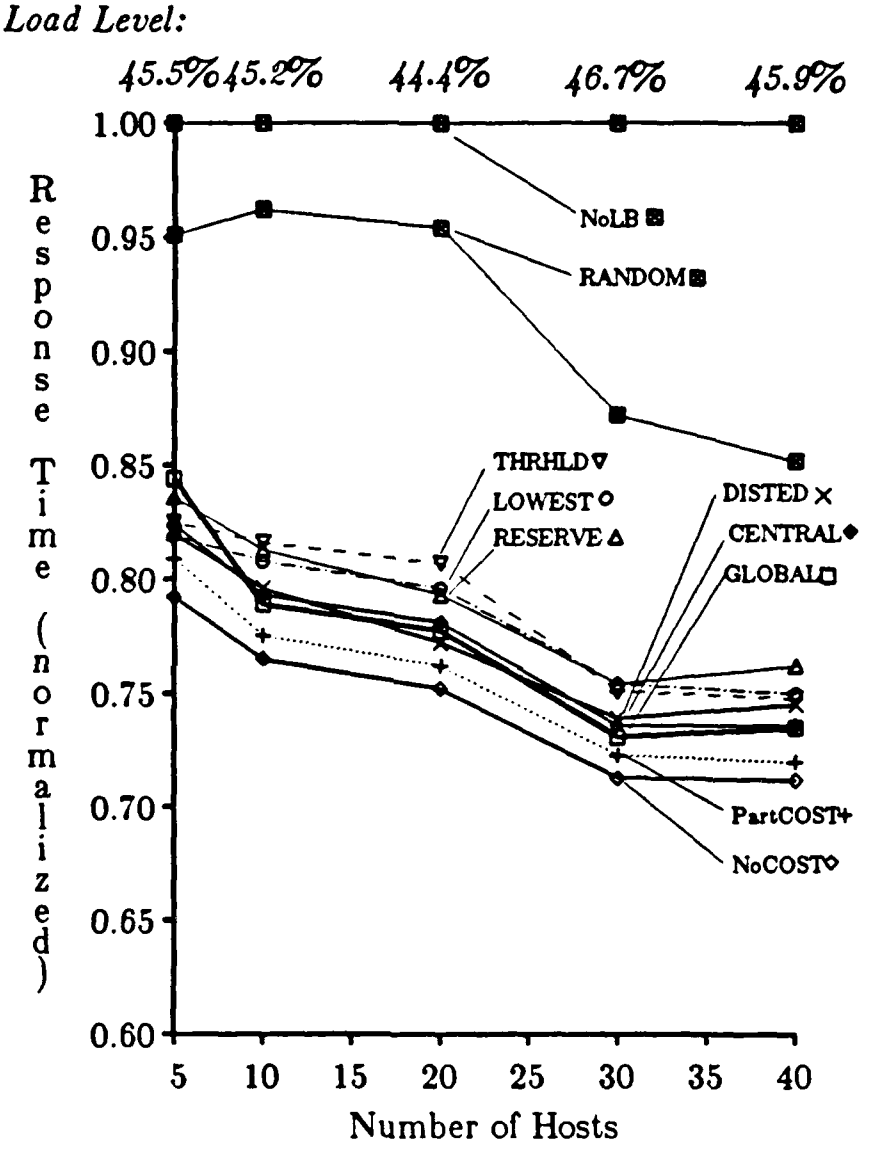

Figure 5.12. Average response times with different system sizes (LBL trace; normalized against the NoLB case).

basic conclusions remained unchanged. Load balancing was found to produce performance improvements in all cases. The algorithms behaved similarly in all three sets of simulations. The effects on load balancing of the immobility factor,

and those of load balancing on individual hosts and job types, were observed to be similar in all three environments.

Chapter 6

Final Remarks

6.1. Conclusions

In the previous chapters, dynamic load balancing has been studied in a number of different computing environments, using a combination of trace-driven simulation, experimental implementation, and measurements. While great caution is called for in attempting to generalize the observations made in case studies, the agreements found in the results from different systems and/or using different research approaches lead us to the belief that many of our findings are not merely properties of a particular system, but rather have more general applicability.

At the beginning of this dissertation, we posed a number of open questions in load balancing, and proceeded to seek answers. Below, we summarize our findings by revisiting these questions.

1) *How much performance improvement can be expected from load balancing?*

   In a local, loosely-coupled distributed system, load balancing using initial job placements alone can improve performance significantly. In the simulation studies of Chapters 3 and 5, and the measurements of Chapter 4, reduction of 30-55% in the average job response time was observed. The workloads used had moderate intensities, with some imbalances; hence, when the workload is heavy and/or more unbalanced, greater improvements may be expected. The reduction is around 20-25% with the LBL data in Chapter 5, due to significantly lower resource contention and queueing than in all the other systems. In all cases, the response times become more predictable. Assuming that a reasonably fast local area network environment is available, measurements show that the costs of load information exchange and job transfer are low.

2) *What algorithm to use?*

   While it is impossible to study all the possible algorithms, we conducted a comparative study of seven algorithms belonging to two families: those

relying on system-wide, periodic load exchange and job placement, and those using host subset probing and placement. We observed a fundamental tradeoff between the amount of overhead incurred and the quality of the job placements. The two families of algorithms were found to provide similar performances in systems with up to 50 hosts. Because of the significant but limited economies of scale in load balancing, there is not much incentive to do load balancing in a very large system, and algorithms with scalability of a few tens to a few hundreds of hosts appear to be sufficient. For the periodic algorithms, using a central agent to collect load information and/or to make placements reduces overhead and improves performance. The performances of some of the implementable algorithms were found to be close to that obtainable without any overhead, thus suggesting that there is probably not much room for further performance improvement unless more load and job information is available and used. It is noteworthy that the above conclusions are common to all the systems we studied.

3) *What load index to use?*

A comparative study of load indices using measurements show that the performance of load balancing is strongly dependent upon the load index used. For the two families of load indices we examined, one based on resource queue lengths, and the other one on resource utilizations, the former was found to be able to reflect the current system load more accurately, thereby producing better performance. Due to the fluctuations in load, smoothing of the instantaneous load values produces further improvement, provided that the averaging interval is not too long as to obscure the current loading conditions.

4) *How does the system's workload affect load balancing performance?*

Generally speaking, load balancing yields greater performance improvement when the workload is heavy and unbalanced. Load balancing is more effective when there are fewer very large jobs, because there are more opportunities to redistribute the workload. This is partly why we observed greater improvement in simulation of the Berkeley and Bell systems than in simulation of the LBL system, and in measurement. We found that load balancing can tolerate immobile jobs to a large extent without suffering significant performance degradation. This is because only a small fraction of the jobs need to be transferred to achieve load balancing effects.

5) *What is the impact of load balancing on system behavior?*

The impact of load balancing on individual hosts and job types is found to be quite uniform using both simulation and measurement. While heavily loaded hosts see big improvements, only slight increases in average response time, or even decreases, are observed on those hosts originally with light loads. Although this is dependent on the degree of system load imbalance,

the beneficial effects of load balancing upon each host seem to be general. Similarly, no drastic differences are observed between the treatments of different types of jobs --- jobs executed remotely or locally enjoy comparable improvements in response time, and the improvement in big jobs is not achieved at the expense of small jobs. While for the algorithms we studied the long term loads of the hosts are made quite even by load balancing, temporal fluctuations are reduced by it but still exist. This is more significant in measurements where multiple processes of the same job are transferred to the same host, inevitably causing a substantial increase in the host's load.

System instability in the form of host overloading is possible, but can be alleviated by using up-to-date load information, and by limiting the jobs eligible for transfer. Also, slight host overloading does not cause much performance degradation.

Our implementation work in UNIX environments shows that general-purpose load balancing can be implemented transparently, and with little change to the system or application software. While we note that very few load balancers are currently in operation, more production quality load balancing systems are called for because the feasibility and performance benefits of load balancing have been clearly demonstrated.

6.2. Future Work

As pointed out in Chapter 1, load balancing is a research topic with many dimensions, and involves a large number of research issues. We studied some of the problems which we believe to be of fundamental importance. Our work may be extended in a number of directions. First, we treated load balancing mainly as a performance issue, while in reality there are many other aspects. Due to the extensive sharing of computing resources required by load balancing, the rigid boundary between hosts is broken, making administration and accounting problems more complex. The security and protection policies of the hosts may be threatened by the presence of foreign processes executing and accessing local resources. Dannenberg proposed, in an environment of personal workstations, the use of dedicated servers, called *butlers*, to monitor the programs running on the host, and to enforce the sharing and protection rules specified by the owner of the station by means of a *policy database* [Dannenberg82]. This provides the workstation with a specifiable level of autonomy, while still allowing sharing among the hosts. We believe that similar functions can be suitably implemented in the LIM and LBM of our load balancer. The open research problems, however, are those concerned with the types of policies suitable for different types of computing environments, e.g., time-sharing systems, banks of compute servers, and networks of personal workstations.

Throughout this research, we assumed the system to be homogeneous, i.e., consisting of hosts of the same type and power. While this base case should clearly be carefully studied and understood, heterogeneous systems offer new opportunities and present new problems to study. We can classify heterogeneous systems into two types. Type A has hosts of varying capacities, but a uniform and compatible software interface, e.g., machines of the same architecture and operating system, but different hardware implementations. Type B has hosts with different hardware architecture and/or operating system. With a Type A heterogeneous system, executing programs remotely and transparently is still feasible, and we conjecture that the load balancing problem is mainly a load index problem. That is, a more general form of load index should be used to reflect the varying capacities of the hosts. The family of indices based on resource queue lengths may be extended naturally to deal with Type A heterogeneous systems. The presence of powerful, yet functionally identical hosts as compute servers makes the computing environment more usable. In Type B systems, load balancing becomes much more difficult. Different, but functionally compatible programs have to be present on each type of host, and conversion mechanisms may be needed, requiring software development and increasing overhead at run time.

The performance of load balancing is closely related to the amount and accuracy of the load and job information made available to the placement decision makers, and the efficiency with which such information is used. We have made minimal assumptions about, and use of, such information (e.g., the job name and resource queue lengths). It can be conjectured that, as more knowledge is gained about job and workload characterization, better predictions about the job resource consumptions and more accurate measurements of system load will be possible, thus making better load balancing possible. Much further work is called for in this area.

Load balancing can be realized through process migration, as well as through initial job placement. Although some successful experiences exist with implementing process migration [Douglis87] [Almes85] [Powell83] [Theimer85], it proves to be difficult to implement cleanly in a production system, mainly due to the persistent residual dependency of the migrated jobs on the originating host. Integrating migration facilities into an existing system is even harder. Consequently, we believe that initial job placement is still the primary means of load balancing, given its simplicity and demonstrated potential for performance improvement. This is not to say, however, that process migration should not be pursued as an alternative means for balancing loads. Leland and Ott performed an interesting comparative study of initial job placement and process migration, and found that the latter can provide some further reduction in mean job response time [Leland86]. Much more work is obviously needed on this topic as well.

We treated the units of execution to be transferred as sequential jobs. Although this is still the predominant case, parallel computation is expected to become much more popular in the future. In a parallel program, several modules are dispatched to different hosts/CPUs for execution. As a result, the execution time of the program may be reduced. We claim that load balancing is still, if not more, important in such an environment. While the modules to be executed in parallel may in some cases be made approximately equal in size (i.e., in the amounts of time they will require to complete), unbalanced loads on the hosts/CPUs may cause one of them to take a longer time to complete, thus becoming the bottleneck of the entire program, and possibly undermining the potential advantage of parallel execution. The modules of the same program may communicate with each other to carry out their tasks. The specific communication requirements, and the resulting overhead, may make it advantageous to consider them together in placement. Interprocess communication aspects raise, of course, the need for new load balancing algorithms that will take them into account.

Bibliography

[Agrawal85]
R. Agrawal and A. Ezzat, "Processor Sharing in NEST: A Network of Computer Workstations," 1st Inter. Conf. on Computer Workstations, November 1985, pp. 198-208.

[Agrawala81]
A. Agrawala and S. Tripathi, "On the Optimality of Semidynamic Deterministic Routing Schemes," Inform. Processing Letters, 13, 1 (October 1981), pp. 20-22.

[Almes85]
G. Almes, A. Black, E. Lazowska, and J. Noe, "The Eden System: A Technical Review," IEEE Trans. Softw. Eng., SE-11, 1 (January 1985), pp. 43-59.

[Alonso86]
R. Alonso, "Query Optimization in Distributed Databases through Load Balancing," Ph.D. Thesis, University of California, Berkeley, June 1986, also as Tech Report UCB/CSD 86/296

[Barak84]
A. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," Tech. Report, Department of Computer Science, Hebrew University of Jerusalem, 1984.

[Beals87]
E. Beals, *Private communication*, May 1987.

[Bershad85]
B. Bershad, "Load Balancing with Maitre d'," Tech Report, UCB/CSD 85/276, Computer Science Division, University of California, Berkeley, December 1985.

[Bokhari79]
S. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," IEEE Trans. Softw. Eng., SE-5, 4 (July 1979), pp. 341-349.

[Bryant81]
R. Bryant and R. Finkel, "A Stable Distributed Scheduling Algorithm," Proc. Inter. Conf. on Distributed Processing Systems, 1981, pp. 314-323.

[Cabrera84]
L. Cabrera, E. Hunter, M. Karels, and D. Mosher, "A User Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX," Tech. Report, UCB/CSD 84/216, Computer Science Division, Univ. of Calif., Berkeley, December 1984.

[Cabrera86]
L. Cabrera, "The Influence of Workload on Load Balancing Strategies," Proc. 1986 Summer USENIX Conference, Atlanta, Georgia, June 1986, pp. 446-458, also as IBM Research Report RJ5271.

[Carey85]
M. Carey, M. Livny, and H. Lu, "Dynamic Task Allocation in a Distributed Database System," Proc. 5'th Int. Conf. on Distributed Computer Systems, Denver, Colorado, 1985, pp. 282-291.

[Carey86]
M. Carey and H. Lu "Load Balancing in a Locally Distributed Database System," Proc. SIGMOD '86, Washington, D.C., 1986, pp. 108-119.

[Chou82]
T. Chou and J. Abraham, "Load Balancing in Distributed Systems," IEEE Trans. Softw. Eng., SE-8, 4 (July 1982), pp. 401-412.

[Chow77]
Y. Chow and W. Kohler, "Dynamic Load Balancing in Homogeneous Two Processor Distributed Systems," Computer Performance, Chandy and Reiser, editors, North Holland, 1977.

[Chow79]
Y. Chow and W. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," IEEE Trans. Computer, C-28, 5 (May 1979), pp. 356-361.

[Chu80]
W. Chu, L. Holloway, M. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," IEEE Computer, 13, 11 (November 1980), pp. 57-69.

[Dannenberg82]
R. Dannenberg, "Resource Sharing in a Network of Personal Computers," PhD Thesis, Carnegie-Mellon University, 1982, also as Technical Report CMU-CS-82-152, Department of Computer Science.

[Douglis87]
F. Douglis, "Process Migration in the Sprite Operating System," *To appear*, Proc. 7th Inter. Conf. Dist. Computing Sys., Berlin, September, 1987, also as Tech. Report, UCB/CSD 87/336, Computer Science Division, Univ. of Calif., Berkeley, January 1987.

[Eager86a]
D. Eager, E. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender Initiated Dynamic Load Sharing," Performance Evaluation, 6, 1 (April 1986), pp. 53-68.

[Eager86b]
D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," IEEE Trans. Softw. Eng., SE-12, 5 (May 1986), pp. 662-675.

[Ezzat86]
A. Ezzat, "Load Balancing in NEST: A Network of Workstations," Proc. 1986 Fall Joint Computer Conference, Dallas, Texas, November 4-6, 1986, pp. 1138-1149.

[Ferrari78]
D. Ferrari, "Computer Systems Performance Evaluation," Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Ferrari85]
D. Ferrari, "A Study of Load Indices for Load Balancing Schemes" Tech. Report, UCB/CSD 85/262, Computer Science Division, Univ. of Calif., Berkeley, October 1985; republished, G. Serazzi, Ed., "Workload Characterization of Computer Systems and Computer Networks", North-Holland, Amsterdam, 1986.

[Ferrari86]
D. Ferrari and S. Zhou, "A Load Index for Dynamic Load Balancing," Proc. 1986 Fall Joint Computer Conference, Dallas, Texas, November 4-6, 1986, pp. 684-690.

[Ferrari87]
D. Ferrari and S. Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications," *To appear*, Proc. PERFORMANCE 87, Brussels, Belgium, December, 1987, also as Tech. Report, UCB/CSD 87/353, Computer Science Division, Univ. of Calif., Berkeley, May 1987.

[Gao84]
C. Gao, J. Liu, and M. Railey, "Load Balancing Algorithms in Homogeneous Distributed Systems," Proc. 1984 Inter. Conf. on Parallel Processing, August 1984, pp. 302-306.

[Hac86]
A. Hac and T. Johnson, "A Study of Dynamic Load Balancing in a Distributed System," Proc. ACM SIGCOMM Symp. on Communications, Architectures and Protocols, Stowe, Vermont, August 1986, pp. 348-356.

[Hagmann86]
R. Hagmann, "Process Server: Sharing Processing Power in a Workstation

Environment," Proc. 6th Inter. Conf. Dist. Computing Sys., Cambridge, Mass., May, 1986, pp. 260-267.

[Harbus86]
R. Harbus, "Dynamic Process Migration: to Migrate or not to Migrate," MS Report, also Technical Note CSRI-42, University of Toronto, July 1986.

[Hsu86]
C. Hsu and J. Liu, "Dynamic Load Balancing Algorithms in Homogeneous Distributed Systems," Proc. 6th Inter. Conf. Dist. Computing Sys., Cambridge, MA, May, 1986.

[Hua85]
K. Hua, "Allocation of Processes and Files for Load Balancing in Distributed Systems," PhD dissertation, Computer Science Division, University of California, Berkeley, October 1985.

[Hwang82]
K. Hwang, W. Croft; G. Goble, B. Wah, F. Briggs, W. Simmons, and C. Coates, "A UNIX Based Local Computer Network with Load Balancing," IEEE Computer, 15, 4 (April 1982), pp. 55-66.

[Johnston86]
W. Johnston and D. Hall, "UNIX Based Distributed Printing in a Diverse Environment," Proc. 1986 Summer USENIX Conference, Atlanta, Georga, June 1986, pp. 514-528.

[Joy80]
W. Joy, "An Introduction to the C Shell," Computer Science Division, University of California, Berkeley, November 1980.

[Joy83]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," Computer Systems Research Group, University of California, Berkeley, July 1983.

[Kratzer80]
A. Kratzer and D. Hammerstrom, "A Study of Load Leveling," IEEE COMPCON, September 1980, pp. 647-654.

[Krueger84]
P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm," CS Dept. Report 539, University of Wisconsin, Madison, April 1984.

[Lazowska86]
E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel, "File Access Performance of Diskless Workstations," ACM Trans. on Computer Systems, 4, 3 (August 1986), pp. 238-268.

[Lee84]
K. Lee and D. Towsley, "A Comparison of Decentralized Load Balancing

Policies in Distributed Systems Characterized by Bursty Job Arrivals," Proc. ACM SIGMETRICS Conf., Raleigh, NC, May 1986, pp. 70-77.

[Leland86]
W. Leland and T. Ott, "Load Balancing Heuristics and Process Behavior," Proc. ACM SIGMETRICS Conf., May 1986, pp. 54-69.

[Lionel85]
M. Lionel and K. Hwang, "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," IEEE Trans. Softw. Eng., SE-11, 5 (May 1985), pp. 491-496.

[Livny82]
M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," Proc. ACM Computer Network Performance Symp., April 1982, pp. 47-55.

[Livny84]
M. Livny, "The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems," PhD Dissertation, Weizmann Institute of Science, Rehovot, Israel, December 1984, also as CS Dept. Report 570, University of Wisconsin, Madison.

[Ma82]
P. Ma, E. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," IEEE Trans. Computer, C-31, 1 (January 1982), pp. 41-47.

[McKusick85]
K. McKusick, M. Karels, and S. Leffler, "Performance Improvements and Functional Enhancements in 4.3 BSD," Proc. Summer USENIX Conference, June 1985, Portland, OR, pp. 519-531.

[Mutka87]
M. Mutka and M. Livny, "Profiling Workstation's Available Capacity for Remote Execution," CS Dept. Report 697, University of Wisconsin, Madison, May 1987.

[Nelson87]
M. Nelson, B. Welch, and J. Ousterhout, "Cacheing in the Sprite Network File System," Tech. Report, UCB/CSD 87/345, Computer Science Division, Univ. of Calif., Berkeley, March 1987.

[Ni81a]
L. Ni and K. Abani, "Nonpreemptive Load Balancing in a Class of Local Area Networks," Proc. IEEE Computer Networking Symp., December 1981, pp. 113-118.

[Ni81b]
L. Ni and K. Hwang, "Optimal Load Balancing Strategies for a Multiple

Processor System," Proc. 1981 Inter. Conf. on Parallel Processing, August 1981, Columbus, Ohio, pp. 352-357.

[Ni85]
L. Ni, C.-W. Zu, and T. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," IEEE Trans. Softw. Eng., SE-11, 10 (October 1985), pp. 1153-1161.

[Powell83]
M. Powell and B. Miller, "Process Migration in DEMOS/MP," Proceedings of the 9th ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983, pp. 110-119.

[Rao79]
G. Rao, H. Stone, and T. Hu, "Assignment of Tasks in a Distributed Processor System with Limited Memory," IEEE Trans. Computer, C-28, 4 (April 1979), pp. 291-299.

[Sherman72]
S. Sherman, F. Baskett III, and J. Browne, "Trace Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System," Comm. ACM, 15, 12 (December 1972), pp. 1063-1069.

[Stankovic84]
J. Stankovic, "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," Computer Networks 8, 1984, pp. 199-217.

[Stankovic85]
J. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," IEEE Trans. Computer, C-34, 2 (February 1985), pp. 117-130.

[Stone77]
H. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Trans. Softw. Eng., SE-3, 1 (January 1977), pp. 85-93.

[Stone78]
H. Stone, "Critical Load Factors in Two Processor Distributed Systems," IEEE Trans. Softw. Eng., SE-4, 3 (May 1978), pp. 254-258.

[Tantawi85]
A. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," Journal ACM, 32, 2 (April 1985).

[Theimer85]
M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," Proceedings of the 10th ACM Symposium on Operating Systems Principles, Orcas Island, Washington, December 1985, pp. 2-12.

[Walker83]
B. Walker, G. Popek, R. English, C. Kline, and G. Theil, "The LOCUS Distributed Operating System," Proceedings of the 9th ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983, pp. 49-70.

[Wang85]
Y. Wang and R. Morris, "Load Balancing in Distributed Systems," IEEE Trans. Computer, C-34, 3 (March 1985), pp. 204-217.

[Wilkes71]
M. Wilkes, "Automatic Load Adjustment in Time-Sharing Systems," Proc. 1971 Harvard Workshop on System Performance Evaluation, Cambridge, Mass., 1971, pp. 308-320.

[Wolff87]
R. Wolff, "Stochastic Modeling and the Theory of Queues," Prentice-Hall, Englewood Cliffs, NJ, forthcoming, 1987.

[Wu80]
S. Wu and M. Liu, "Assignment of Tasks and Resources for Distributed Processing," Proc. COMPCON, Fall 1980, pp. 655-662.

[Yu86]
P. Yu, S. Balsamo, and Y. Lee, "Dynamic Load Sharing in Distributed Database Systems," Proc. 1986 Fall Joint Computer Conference, Dallas, TX, November 1986, pp. 675-683.

[Yum81]
T. Yum, "The Design and Analysis of a Semidynamic Deterministic Routing Rule," IEEE Trans. Commun., COM-29 (April 1981), pp. 498-504.

[Zatti85]
S. Zatti, "A Multivariable Information Scheme to Balance the Load in a Distributed System," Master Report, also Tech. Report, UCB/CSD 85/234, Computer Science Division, Univ. of Calif., Berkeley, May 1985.

[Zhou85]
S. Zhou, H. Da Costa, and A. J. Smith, "A File System Tracing Package for Berkeley UNIX," Proc. Summer USENIX Conference, Portland, OR, June 11-14, 1985, pp. 407-419.

[Zhou86]
S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," Tech. Report, UCB/CSD 87/305, Computer Science Division, Univ. of Calif., Berkeley, September 1986, also submitted for publication.

[Zhou87a]
S. Zhou, "An Experimental Assessment of Resource Queue Lengths as Load Indices," Proc. Winter USENIX Conference, Washington, D.C., January

1987, pp. 73-82, also as Tech. Report, UCB/CSD 85/298, Computer Science Division, Univ. of Calif., Berkeley, April 1986.

[Zhou87b]
S. Zhou and D. Ferrari, "An Experimental Study of Load Balancing Performance," *To appear*, Proc. 7th Inter. Conf. Dist. Computing Sys., Berlin, September, 1987, also as Tech. Report, UCB/CSD 87/336, Computer Science Division, Univ. of Calif., Berkeley, January 1987.

[Zhou87c]
S. Zhou and R. Zicari, "Object Management in Local Distributed Systems," Tech. Report, UCB/CSD 86/267, Computer Science Division, September 1985. *to appear*, Journal of Systems and Softw., North-Holland.